Macintosh® Introduction to MacApp 2.0 and Object-Oriented Programming

Draft

April 14, 1989

Developer Technical Publications

© Apple Computer, Inc. 1989

APPLE COMPUTER, INC.

This manual and the software described in it are copyrighted, with all rights reserved. Under the copyright laws, this manual or the software may not be copied, in whole or part, without written consent of Apple, except in the normal use of the software or to make a backup copy of the software. The same proprietary and copyright notices must be affixed to any permitted copies as were affixed to the original. This exception does not allow copies to be made for others, whether or not sold, but all of the material purchased (with all backup copies) may be sold, given, or loaned to another person. Under the law, copying includes translating into another language or format.

You may use the software on any computer owned by you, but extra copies cannot be made for this purpose.

The Apple logo is a registered trademark of Apple Computer, Inc. Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

© Apple Computer, Inc., 1989
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Apple, the Apple logo, A/UX, LaserWriter, MacApp, and Macintosh are registered trademarks of Apple Computer, Inc.

APDA, AppleTalk, MPW, Finder, MultiFinder, ResEdit, and Rez are trademarks of Apple Computer, Inc.

Adobe Illustrator is a trademark of Adobe Systems Incorporated.

ITC Garamond and ITC Zapf Dingbats are registered trademarks of International Typeface Corporation.

Microsoft is a registered trademark of Microsoft Corporation.

POSTSCRIPT is a registered trademark of Adobe Systems Incorporated.

UNIX is a registered trademark of AT&T Information Systems.

Simultaneously published in the United States and Canada.

Contents

Figures and tables ix

Preface xiii

Prerequisites xv
- Hardware xv
- Software xv
- What you should already know xvi

Other books you should have xvii
What this manual contains xvii
How to use this manual xviii
Nomenclature and typographic conventions xix
Conventions in the MacApp code xix
Graphical conventions xx
Where to get information xxii
- About APDA xxii
- About Developer Programs xxii

1 Why MacApp? 1

What is MacApp? 3
Programming without MacApp 4
Programming with MacApp 5
- The division of labor 5

The benefits of MacApp 7

2 An Introduction to Object-Oriented Programming 9

The big picture 11
- Record types and record variables 12
- Classes and Instances 13
- Flow of control in object-oriented programs 17

Classes 27
- Inheritance 28
- Method definitions 31
- Override methods 32

Objects 34
- An instance method 35
- Inheritance and override methods 37
  - Override methods that don't call their inherited method 37
  - Override methods that do call their inherited method 38
- Privacy between instances 41

Ramifications of object-oriented programming 42
- Ease of expansion 42
- Implementing standard features with MacApp 42
- Customization without tears 43

3 Object Pascal 45

Object Pascal file organization 47

Class definitions 51
- Field declarations 52
- Method declarations 52
- Override method declarations 54

Instances 55
- Record handles and record instantiations 55
  - Instantiation through record variables 55
  - Instantiation through record pointers 55
  - Instantiation through record handles 56
- Object references 56
- Method call syntax 57

Method definitions 58
The SELF keyword 59
Methods that call other methods of the calling instance 61
Methods that call methods of other instances 61
Calling a method of a global object 61
Calling a method of a local object 62
Calling a method of an instance linked by an object reference field 62
Override methods 67
Override methods that call the inherited method 68
More about Object Pascal 69
Creating and freeing instances 69
Initialization methods 69
Privacy between instances 72
Object reference variables 72
Summary of Object Pascal syntax 73

4 Introduction to MacApp: Organization 75
Source code: Organization of the files 77
Ancestry: Organization of the classes 79
Higher-level classes 82
Subclasses of TView 84
Summary of the ancestry tree 85
Architecture: Organization of the concepts 86
Storing data 86
Displaying information 88
Hierarchy of views 88
Windows 93
Scrollers 94
Main content 95
Palettes 95
Scroll bars 95
Other specialized views 96
Executing commands 98
Summary 101

5 Introduction to MacApp: Flow of Control 103
Event handling in MacApp 105
Overview of MacApp 106

Assigning events to instances 111
Target chains 115
The command chain 115
How the chain is built 117
How the head of the chain is determined 118
How the chain is traversed 119
The click chain 123
How the chain is built 123
How the head of the chain is determined 123
How the chain is traversed 123
The cohandler chain 125
How the chain is built 125
How the head of the chain is determined 125
How the chain is traversed 126
Summary 127

6 How to Use MacApp 129

Installing MacApp 131
File naming conventions 132
Building a MacApp program 133
The architecture of MacApp's build system 133
MABuild's input and output 135
Inputs 135
Outputs 136
The simplest builds 137
Other builds 138
Creating a makefile 139
Options for MABuild 141
Object folders 144
MPW shell variables for MABuild 147
MacApp Rez files 148
Include files 148
The 'cmnu' resource type 149
MBAR' resources 149
The structure of a MacApp program 150

7 MacApp Debugging Facilities 153

MacApp's debugging gear 155
What controls debugging code 156
Compiler variables 156
The $D switches 157
Including debugging code 158
The Debug window 159
Application mode and debug mode 159
Using WriteLn statements with the Debug window 159
Reading debugging information in application mode 160
The Debug menu 160
New Inspector Window 160
Make Front Window Modal 160
Do First Click For This Window 161
Refresh Front Window 161
Allow Trace Of Menu Setups 161
Allow Trace During Idle 161
Enter MacApp Debugger 161
Show Debug Window 162
Show Software Version 162
Inspector windows 162
Using Inspector windows 162
The Fields method 164
The GetInspectorName method 165
The Interactive Debugger 166
Entering debug mode 166
What the Interactive Debugger prints when it starts 168
Using the Debug window in debug mode 169
Interactive debugger commands 169
The Help/Status command (?) 170
The Recent History command (R) 170
The Parameters command (P) 172
The Locals command (L) 173
The Fields command (F) 173
The Display Memory command (D) 174
The Stack Crawl command (S) 175
The Display More command (M) 175

The Trace command (T) 175
The Single Step command (Space bar) 175
The Go command (G) 176
The Breakpoint command (B) 176
The Clear Break command (C) 176
The Output Redirect command (O) 176
The Quit command (Q) 177
The Heap and Stack command (H) 177
The Toggle Flag command (X) 178
The Inspect Object command (I) 180
The Enter MacsBug command (E) 181
Using MacsBug with MacApp 181

Appendix A Changes Since MacApp 1.1 183
Changes to the architecture 185
Views 185
The old display architecture 185
The new display architecture 187
The most important classes in the new architecture 189
Large coordinates 190
Text 190
Dialog boxes 190
Grids 194
Inspector 195
AppleTalk 196
MultiFinder support 196

Index 197

Figures and tables

Preface About This Manual ix

Figure P-1 Conventions in flowcharts xxi

1 Why MacApp? 1

Figure 1-1 Components of a window 6

2 An Introduction to Object-Oriented Programming 9

Figure 2-1 Tree-like structure based on routines 11
Figure 2-2 A record type 12
Figure 2-3 A record type and record instances 13
Figure 2-4 A class and two instances with no methods 14
Figure 2-5 A class with a method definition 15
Figure 2-6 A method as it is defined and as it executes 16
Figure 2-7 An expanded Company class 17
Figure 2-8 Instances referring to each other 18
Figure 2-9 The completely defined Company class 19
Figure 2-10 A method sending a message to the calling instance 20
Figure 2-11 A method sending a message to a different instance 21
Figure 2-12 Objects exist in memory 23
Figure 2-13 Objects have been initialized 23
Figure 2-14 Company A's printing method is called 24
Figure 2-15 Company B's printing method is called, and the program is finished 25
Figure 2-16 An Employee class 28
Figure 2-17 A simple hierarchy of classes 29
Figure 2-18 Multiple levels of inheritance 30
Figure 2-19 The generic method definition of the Print method of the Employee class 32
Figure 2-20 An override method that does not call its inherited method 33
Figure 2-21 An override method that calls the inherited method 34
Figure 2-22 The four Employee instances 35
Figure 2-23 The Print method definition and a specific object 36
Figure 2-24 The WeeklyPaycheck methods of the Hourly and Salaried classes 38

Figure 2-25 A method that calls an inherited method 39
Figure 2-26 More examples of override methods called by specific instances 40

3 Object Pascal 49

Figure 3-1 Typical file organization of an Object Pascal program 49
Figure 3-2 Possible file organization of a large Object Pascal program 50
Figure 3-3 A method as it is defined and as it executes 60
Figure 3-4 Uninitialized objects in memory 65
Figure 3-5 Object references and objects in memory 66

4 Introduction to MacApp: Organization 81

Figure 4-1 The ancestry tree for the classes in MacApp 80
Figure 4-2 Tree of views 89
Figure 4-3 A fairly complex window with one scroll bar and a palette 90
Figure 4-4 Exploded version of window in Figure 4-3 91
Figure 4-5 View tree for window in Figure 4-3 92
Figure 4-6 Parts of the window 94
Figure 4-7 TEView, TGridView, and TDialogView 97
Figure 4-8 The main event handlers and their responsibilities for commands 99

Table 4-1 MacApp units 77
Table 4-2 Naming conventions for MacApp files 79
Table 4-3 Examples of simple and complex commands 100

5 Introduction to MacApp: Flow of Control 107

Figure 5-1 Fundamental overview of a MacApp application 107
Figure 5-2 Flow of control: MainEventLoop 108
Figure 5-3 The CASE statement in TApplication 109
Figure 5-4 The events most important to you 113
Figure 5-5 Target chain from View B1c to the application object 116
Figure 5-6 Diagram of fNextHandlers 118
Figure 5-7 Possible command chain when the user chooses Quit 120
Figure 5-8 The view tree 124
Figure 5-9 The cohandler chain 125

6 How to Use MacApp 133

Figure 6-1 The MacApp 2.0 Build system 134

Table 6-1 Configuration for building an application without a makefile 137
Table 6-2 Make variables 140

Table 6-3 MABuild options 142
Table 6-4 Two-letter compiler option mnemonics 146

7 MacApp Debugging Facilities 245

Figure 7-1 A sample session with an Inspector window 163
Figure 7-2 Output from the Recent History command 171
Figure 7-3 Output from the Fields command 174

Table 7-1 Compile-time debug flags 156

Appendix A Changes Since MacApp 1.1 183

Figure A-1 The display architecture from MacApp 1.1 186
Figure A-2 Tree of views 188
Figure A-3 View tree for a MacApp window 190
Figure A-4 The classes defined in UDialog 193
Figure A-5 An Inspector window 195

Preface

THE MACAPP® OBJECT-ORIENTED APPLICATION FRAMEWORK is a set of libraries you can use to speed your software development process and to create more robust applications for Macintosh® computers. This manual is an introductory volume to a multi-volume suite that includes introductions to object-oriented programming and MacApp, a tutorial, a cookbook, a general reference, and an encyclopedic reference to procedures, functions, methods, and routines in MacApp. Most of the reference material is not included in this volume; however, this book is a complete (if not comprehensive) introduction to MacApp and object-oriented programming. ■

Prerequisites

To use MacApp, you must have certain hardware and software; to make good use of this manual, you must have had some experience programming.

Hardware

You must have a Macintosh computer with 128K ROMs, a hard disk, and at least one megabyte of memory to develop an application with MacApp; however, most applications of average size will require you to have at least two megabytes of memory while you are developing them.

Using MacApp, it is possible to create applications that can run on most computers in the Apple® Macintosh line, including the Macintosh Plus, the Macintosh SE, the Macintosh SE/30, the Macintosh II, the Macintosh IIcx and the Macintosh IIx. To support all of these different flavors of Mac, your application should check to make sure its memory requirements do not exceed the memory limits of the computer it is running on, and it should also check for features that are not supported across the product line, such as color screen display or support for multiple monitors. Additional information on including this kind of checking in your code is detailed in the Macintosh Tech Note "Checking for Specific Functionality," available from APDA at the address listed at the end of this Preface.

If you want your application to support the 64K ROM in the original Macintosh, you must provide the necessary code to do so, such as implementing the MFS file system used prior to the introduction of the hierarchichal file system (HFS) in the 128K ROM. MacApp 2.0 does not support the Macintosh XL; if you want your application to run on this machine, you must provide all necessary code.

Software

MacApp 2.0 works under the MPW™ (Macintosh Programmer's Workshop) development system version 3.0 and comes with an interface for MPW Pascal.

What you should already know

If you are a competent Pascal or C programmer and you have played with a Macintosh enough to have a feeling for what makes a good application, then you know enough to use this manual. You don't have to know anything about object-oriented programming, and you don't have to have written a single line of code for a Macintosh program.

However, if you have never programmed a Macintosh before, you can do some reading to make learning MacApp much easier. The books listed here appear in the order of how vital they are to your understanding:

- Before you begin with this manual, read Chapters 1, 2, and 3 in *Programmer's Introduction to the Macintosh Family* to get an overview of how programming the Macintosh is different from programming other computers. These chapters will also give you a good idea what it's like to program the Macintosh without an aid like MacApp.

  You should go on to read the rest of the *Programmer's Introduction to the Macintosh*, especially Chapter 4 ("Memory Management"), Chapter 5 ("Display and Graphics Routines"), and Chapter 7 ("File Management") as they seem necessary. The information in Chapter 6 ("The User Interface Toolbox") is interesting and important, but it may seem confusing. Wait until you understand MacApp fairly well before you read Chapter 6.
- To get an idea of the different elements you can include in your interface, browse through Apple's *Human Interface Guidelines* and look at the information on extensions to the user interface in *Inside Macintosh.* These were the guidelines by which MacApp was created; you should use them as your guide to create exciting and effective applications.
- After you have finished Chapters 1 through 6 of this book and before you start programming, read the QuickDraw chapter of *Inside Macintosh* to learn how to create images on the Macintosh screen.
- When you are ready to create programs that will write files to disk, read the File Manager chapter of *Inside Macintosh.*
- After you understand the basic concepts of MacApp, read the Resource Manager chapter of *Inside Macintosh.*
- The more you know about MPW, the Macintosh Programmer's Workshop, the more efficiently you will be able to develop applications with MacApp. See the *Macintosh Programmer's Workshop 3.0 Reference.*

These books contain information about Macintosh ROM routines. See "Other books you should have," later in this preface, for bibliographic information on these works.

Because you are using MacApp, you will not need to know the specifics of many of the Macintosh ROM routines detailed in *Inside Macintosh.* MacApp provides a framework that will keep you from having to use many of these routines directly; instead, MacApp calls them for you when appropriate. However, you should look at the information on the User Interface, QuickDraw, the Memory Manager, the Resource Manager, and the File Manager. You may also want to browse through chapters concerning the Window Manager, the Menu Manager, and the Control Manager, as well as any information concerning sound and color. Another document that is recommended reading is the *Programmer's Guide to MultiFinder™* , available from APDA. (APDA ordering information is listed at the end of this Preface.)

Other books you should have

Here is a list of books you should have within reach while programming with MacApp.

- *MPW Assembler Reference* (Apple Computer, 1986). If you intend to include assembly language routines in your applications, you will need this manual.
- *MPW Pascal Reference* (Apple Computer, 1986). This manual documents Apple's version of the Object Pascal language.
- *MPW Reference and MPW Command Reference* (Apple Computer, 1986). These manuals describe the development system you need to use with MacApp. The more familiar you are with MPW, the more efficiently you'll be able to work.
- *Programmer's Introduction to the Macintosh Family* (Addison-Wesley, 1988). This book explains the important concepts you need to know in order to program the Macintosh.

What this manual contains

Chapter 1 introduces the concepts behind MacApp, starting with an overview of MacApp itself. Chapters 2 and 3 introduce object-oriented languages in general, and Pascal in particular. Next, Chapters 4 and 5 describe the physical, hierarchical, and conceptual organization of MacApp and show the flow of control in Macintosh applications, as governed by MacApp. Chapter 6 tells you how to use MacApp's newest build features to create your own applications, and Chapter 7 contains information on using MacApp's interactive debugger.

Appendix A contains information for readers who may be familiar with MacApp 1.1, describing the differences between the old and new versions of MacApp.

How to use this manual

The more you already know about object-oriented programming, programming the Macintosh, or using MacApp, the more of this manual you can skip.

The only readers who may find it useful to read this whole manual are those who already know how to program the Macintosh but are not yet familiar with MacApp or object-oriented programming. Others may want to customize their reading as follows:

If you have never programmed a Macintosh, you should carefully consider the suggestions in the section "What you should already know."

If you are familiar with object-oriented programming, you can skip Chapters 2 and 3.

If you are an impatient learner, you should still read Chapter 1. Then, if you are adventurous, skip to Chapter 6, "How to Use MacApp." If you feel confused, go back and read the rest of the information in Chapters 2–5..

If you are well-acquainted with MacApp 1.1, you may choose to read only Appendix A, "Changes Since MacApp 1.1," and the section in the *MacApp 2.0 General Reference* on "Converting From MacApp 1.1 to MacApp 2.0." Then you will probably have to rely heavily on the *MacApp 2.0 Cookbook* for the first applications you write with MacApp 2.0.

- *Note:* MacApp 1.1.1 included only bug fixes; converting an application written with MacApp 1.1.1 is essentially the same as converting one created with MacApp 1.1.

Nomenclature and typographic conventions

This manual uses `courier` type to represent code fragments. Note that **bold** type is sometimes also used to highlight important concepts, especially at the beginnings of paragraphs.

Like all technical manuals, this manual uses some words in special, conventionalized ways. These words and other conventions are explained in detail in the text, but here is a list for quick reference:

- **Routine** means any procedure or function.
- **Method** means any routine belonging to a class of objects.
- ***Your*** indicates an identifier or part of an identifier that you are expected to replace, as in **TYour*Application***. If your application were named Corporate, you would replace the name as TCorporateApplication.
- **Generic references** to instances of a particular class are given the name of the class minus the "T"—for example, in this manual a generic instance of the class TWindow is just called a window.
- **Pascal reserved words** and predefined type names are written in all capital letters.
- **ROM routines** are routines in the Macintosh ROM and system software. They are documented in *Inside Macintosh.*

Conventions in the MacApp code

- All variable and constant identifiers begin with a lowercase letter.
- Global variable identifiers begin with a lowercase *g* (for example, gTarget).
- Command constants begin with a lowercase *c* (for example, cUndo).
- Most other constants begin with a lowercase *k* (for example, kWantHScrollBar).
- Fields of object classes begin with a lowercase *f* (for example, fDocument).
- All routine names, both object-oriented and otherwise, begin with an uppercase letter (for example, Enable).
- Names of classes begin with an uppercase *T* (for example, TApplication).
- Compile-time variables begin with a lowercase *q* (for example, qDebug).
- Private global variables begin with a lowercase *p* (for example, pInitialized).

MacApp and Object Pascal do not add any special checking to enforce these conventions. Uppercase and lowercase characters are equivalent in Pascal identifiers because the Pascal compiler coverts all characters to uppercase. When you use the MPW Assembler, all identifiers are also converted to uppercase, unless you use the assembler directive to make case significant. By default, MacApp does set this directive to make case significant.

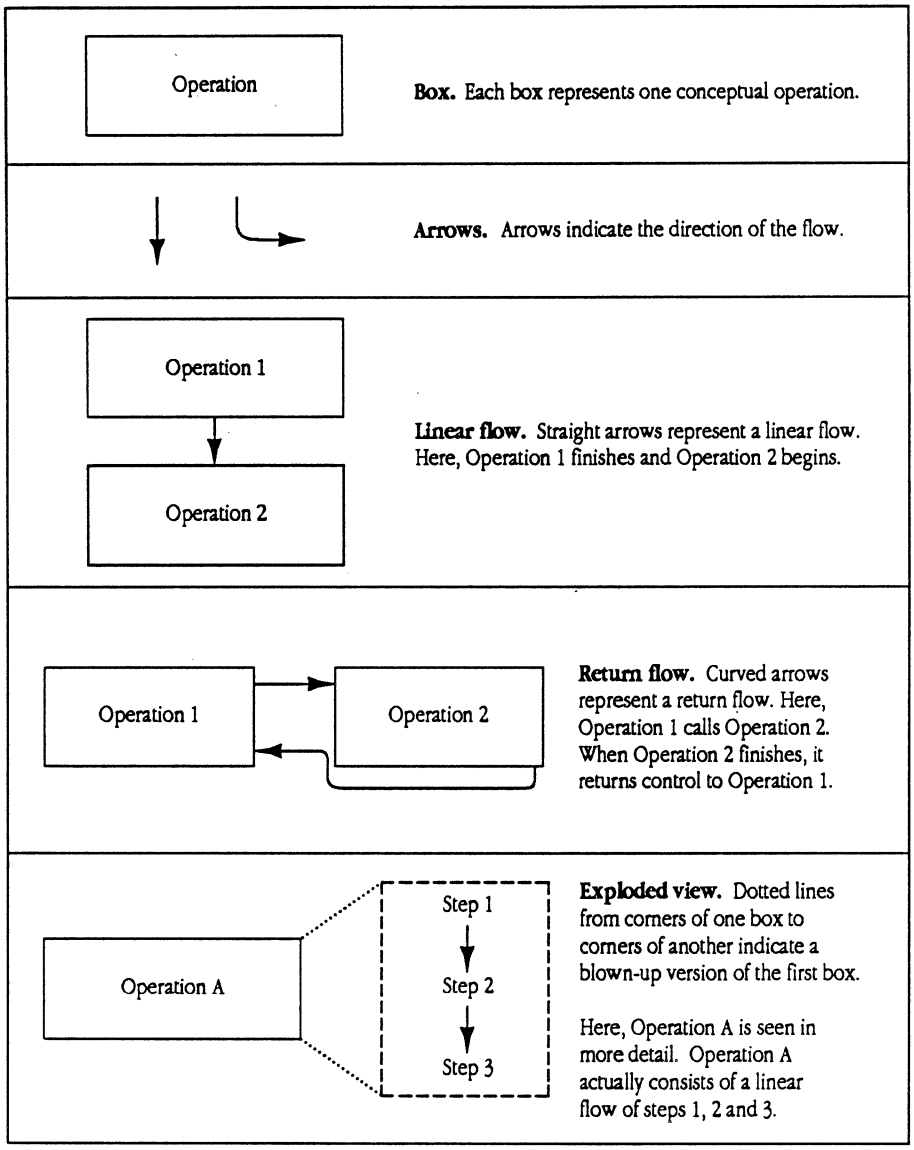
Graphical conventions

Figure P-1 illustrates the graphical conventions used in the diagrams and flowcharts in this book. As in a traditional flowchart, each box can be considered a single operation. All the flowcharts will start at the top, and each box will contain one of five items: the name of an object's method, the name of a global routine, the name of a ROM call, a Pascal statement, or a general description. If there is an asterisk in a box, it means that you can find more information about the indicated routine in the box in some other flowchart. If the box is shaded, it means that the MacApp version of that routine does almost nothing or does not exist, and you should write it or override it if you want it to work.

Solid arrows indicate the direction of flow of control. Hollow arrows are used to show hierarchy of object classes. Dashed arrows indicate pointers to locations in memory.

■ **Figure P-1** Conventions in flowcharts

Graphic conventions


Operation
Box. Each box represents one conceptual operation.
Arrows. Arrows indicate the direction of the flow.
Operation 1
Operation 2
Linear flow. Straight arrows represent a linear flow. Here, Operation 1 finishes and Operation 2 begins.
Operation 1
Operation 2
Return flow. Curved arrows represent a return flow. Here, Operation 1 calls Operation 2. When Operation 2 finishes, it returns control to Operation 1.
Operation A
Step 1
Step 2
Step 3
Exploded view. Dotted lines from corners of one box to corners of another indicate a blown-up version of the first box.
Here, Operation A is seen in more detail. Operation A actually consists of a linear flow of steps 1, 2 and 3.

Where to get information

Addison-Wesley books are available at commercial bookstores. Books and manuals published by Apple are available through APDA™ at the address listed below. Technical notes and other materials of interest to Macintosh application developers are also available from APDA.

About APDA

APDA provides a wide range of technical products and documentation, from Apple and other suppliers, for programmers and developers who work on Apple equipment. (MPW is distributed through APDA.) For information about APDA, contact them as follows.

APDA
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 33–G
Cupertino, CA 95014–6299

Telephone: 1-(800)-282-APDA or 1-(800)-282-2732
Fax: (408)-562-3971
Telex: 171-576
AppleLink: DEV.CHANNELS

About Developer Programs

If you plan to develop hardware or software products for sale through retail channels, you can get valuable support from Apple Developer Programs. Write to them at the following address:

Apple Developer Programs
Apple Computer, Inc.
20525 Mariani Avenue, Mailstop 51–W
Cupertino, CA 95014–6299

Chapter 1 **Why MacApp?**

MACAPP® IS A TOOL THAT HELPS MACINTOSH PROGRAMMERS create better applications in less time. It helps programmers work faster by allowing them to program in a style well suited to Macintosh® applications. It also facilitates the implementation of the standard Macintosh interface, including everything from scrollable, resizable windows to multipage printing and undoable commands. It makes programs better because it provides code carefully written by expert Macintosh programmers; as long as you exercise similar care, MacApp applications are virtually guaranteed to be compatible across the whole Macintosh line and to work with all other Apple® products. ■

What is MacApp?

In technical terms, MacApp is an object-oriented application framework. This phrase—"object-oriented application framework"—contains a lot of information and deserves further explanation. MacApp is *object-oriented* because it uses a style of programming that is organized around **objects** instead of procedures and functions. Chapter 2 explains object-oriented programming and the definition of objects in detail. In MacApp, objects represent Macintosh entities such as windows and dialog boxes as well as real-world entities such as chess pieces, rectangles, and automobiles.

MacApp is an *application framework* because it provides a general structure that you can use to build almost any application. With MacApp and less than one page of your own code, you can have a complete Macintosh application that creates windows, handles mouse clicks, prints files, and supports almost every standard feature Macintosh applications are likely to have. Of course, this generic program will have no specific functionality; that much you will have to do yourself. But when you wish to use the standard features, you will find MacApp does most of the work for you. If you wish to stray a little from these standards, you will have to do a little extra work. In general, the more you want to stray, the more work you will have to do.

In more specific terms, MacApp is a large library of code written in Object Pascal and MPW™ Assembler Language. The library includes global procedures, functions, and methods that provide the standard Macintosh features, and optional units that will provide printing, dialog boxes, and text editing. MacApp also includes logic that will supervise when your own customized code is to be called.

Finally, you should note that MacApp is a framework for *applications* only. MacApp is not the appropriate tool for building other sorts of programs. It cannot be used to create device drivers, desk accessories or HyperCard® XCMDs.

Programming without MacApp

Programming a Macintosh, whether or not you use MacApp, is different from programming most other computers. While programs for other computers constrain the user to one of a few actions, Macintosh applications put the user in charge. Macintosh applications must be ready to handle anything at any time: Without warning, the user can move the mouse, type a number, click in the menu bar, insert a disk, enter a command from the keyboard, or use the network.

Because Macintosh applications give users so much flexibility, they require a different structure than traditional programs. Macintosh application programs are built around a **main event loop** which continually looks for any event that requires a response. When an event comes along, it temporarily passes control to some other piece of the program responsible for handling that particular type of event. The program handles the event and returns control to the main event loop. The process continues until the user quits the application.

While the main event loop itself does not require very much code, the code required to handle all the events is extensive. For instance, each time the mouse button is clicked, the application must determine whether the action requires a response and, if so, what sort of response it merits. A click in the menu bar may cause a menu to appear and will require that the mouse be tracked; if the mouse button is released over a menu item, the application will have to execute the appropriate command. A click in a window might activate the window. A click in a scroll bar might cause the view to change. And a click in a title bar might cause the window to move.

Moreover, applications require a fair amount of code to provide error checking and standard features such as printing, saving, and posing queries through dialog boxes. Much of this work is provided by code in the toolbox of the Macintosh ROM, but not as much as you might think. The toolbox knows nothing of windows with scroll bars, undoable commands, or clusters of radio buttons. It takes a lot of code just to provide these standard features.

Even someone who has no programming experience can tell that features don't come automatically. Most Macintosh users can tell you, in anguished tones, which programs have some crucial command that is not undoable. You can probably think of a program that has windows without scroll bars or that doesn't allow multiple windows to be open at once. If these features were free and easy, every program would have them.

That is the idea behind MacApp. MacApp makes it so easy to include standard features that programmers find it less work to provide the features than not.

Programming with MacApp

If you have programmed much at all, you have probably learned that you can save a lot of time by stealing lines of code from other programs and changing a few parameters here and there. Macintosh application programs have a great amount in common with each other, so borrowing code is particularly effective. However, plagiarizing code is a difficult endeavor, even if it is your own code you are stealing. It is easy to overlook a missing parameter or a line of code that was meaningful only in some other program. MacApp puts the very best of Macintosh programming at your disposal without these pitfalls.

The manner in which MacApp allows you to borrow and customize code relies heavily on the use of object-oriented programming techniques. The next two chapters are devoted to this subject. For now, imagine that MacApp was not written in an object-oriented language, but rather in the procedure-oriented style more familiar to most programmers. In this hypothetical (and false) model, MacApp is a complete template application. It has a main event loop that waits for events and a number of routines that handle them. You just fill in the blanks.

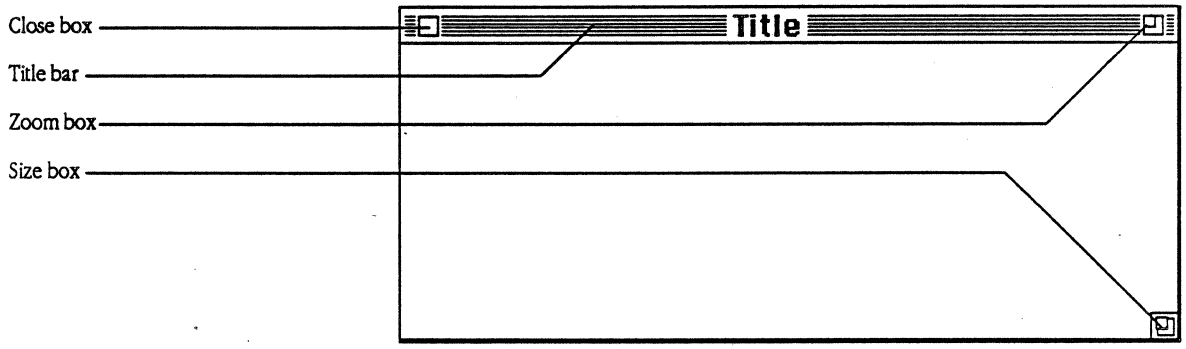
The division of labor

In general, you can assume that MacApp implements the typical ways a Macintosh program should respond to the user's actions. Thus, the portion of MacApp responsible for windows knows how to respond if the user clicks in any of the window's controls, because virtually every application handles these clicks in the same way. If users want to change the size of the window, they always click the size and zoom boxes; if they want to move the window, they always drag the title bar. However, you can't expect MacApp to know what the content of any particular window might be. It is MacApp's job to handle events involving the controls of the window, since these are predictable; it is your job to handle events involving the content of the window, since only you know what this might be.

Commonsense logic of this sort will help you figure out what MacApp does for you and what you must do for yourself. Sometimes MacApp knows *when* something must be done, but not *how*. When this is the case, MacApp will call a specific routine that it expects you to have written. Much of this manual will be devoted to teaching you which routines you have to write and how they should be named. Here are some examples of how the division of responsibility works in MacApp.

- **Managing windows.** Windows can have a variety of standard components, pictured in Figure 1-1.

- **Figure 1-1** Components of a window



MacApp provides some of these components at your discretion. MacApp's code draws the window and responds if the user manipulates the size box, title bar, or zoom box. Your code is responsible for drawing the contents of the window and responding if the user manipulates the contents. In addition, if you want any of these components to work in an unconventional manner, you will have to add some code of your own.

- **Managing menus.** MacApp does most of the work involved in managing most menus. You must inform MacApp of the items that are to be listed in the menus, how they might be chosen from the keyboard, and their state (enabled, checked, and so forth). MacApp will automatically display the menus and highlight the commands in the usual way.
- **Executing menu commands.** Mac App handles all of the commands standard to a Macintosh application such as New, Open, Close, Print, Cut, Copy, and Paste. It also controls the execution of application switching under Multifinder™ via the Apple menu. Since MacApp cannot anticipate commands that are unique to your application, you must write the code for these commands yourself. Additionally, you will need to implement data structures and methods for MacApp to call when executing your application's Save command. If a command is undoable, you will have to write at least three pieces of code: (1) how to execute the command, (2) how to undo it, and (3) how to redo it. MacApp knows when to call which piece of code.
- **Handling errors.** MacApp provides a framework for error handling, as well as a number of routines for detecting and handling errors that fit into that framework. You supplement MacApp's error handling in a variety of ways: by calling MacApp's error handling and detection routines, by providing more specific error messages for those routines, and by providing your own routines.

- **Printing.** MacApp provides standard multipage printing capabilities that you can enable by linking in an extra module and writing three lines of code. If you wish to add any extra functionality, such as drawing borders around the text or placing headings on every page, you must add them yourself, though MacApp has provisions to make this work relatively easy.
- **Editing text.** MacApp provides a text editor that you can access by linking in an extra module and adding another three lines of code. Again, you can enhance this editor by adding your own custom code.
- **Filing documents.** MacApp provides a framework for saving files and checking to make sure sufficient disk space is available. Since the format in which you save the data is not predictable, you must provide the code to read and write data. MacApp knows when to call this code.

This division of responsibility is one of the most difficult concepts in learning how to use MacApp. Even once you learn who does what, you may lose faith that MacApp will do its part, since you are not providing the code yourself. This manual will make a special effort to make your responsibilities clear; you can trust MacApp to do the rest.

The benefits of MacApp

If you develop applications with MacApp, *you* will benefit because MacApp provides you with a superior development system, and the *users* of your application will benefit because the application will work the way they expect. MacApp is written according to Apple's own specifications. As a result, your application is likely to work on any computer in the Macintosh line. It can work on the Macintosh Plus as well as the Macintosh IIx, and it will work with MultiFinder and A/UX®, provided you maintain MacApp's standards of compatibility. Moreover, because MacApp is supported by Apple, your application is likely to continue working with any new Macintosh systems.

Besides adding more functionality to your application and making it easier to maintain, MacApp makes the development cycle more productive. Since MacApp provides the basic framework, programmers can focus on the more interesting portions of an application. As a result, you can speed up development or get more done in the same amount of time. In addition, MacApp allows you to develop the application in an extremely modular fashion. You can recompile after adding each small method and get a testable, working version every time. You will always have a working application to show your boss or your clients. And if there is a problem, MacApp provides you with a high-level object-oriented debugger.

Perhaps the greatest benefit MacApp affords you is the pleasure your users will get from running your new application. Because your application was written with MacApp, the user interface—the window controls, menus, and so on—will be likely to work just as users expect, and they will spend less time learning the application and more time enjoying it. Because your application was written with MacApp, it's more likely that it will work under MultiFinder and with future revisions of the Macintosh Operating System. These are the criteria by which users decide whether an application is truly professional and worth purchasing.

In short, an application written with MacApp is likely to be a better application.

Chapter 2 An Introduction to Object-Oriented Programming

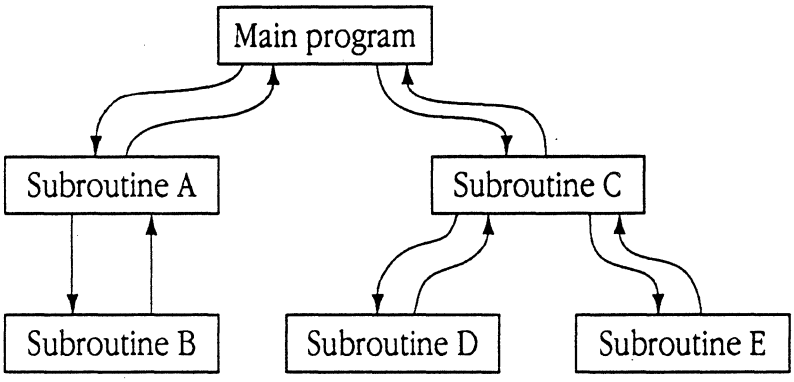
MACAPP IS A POWERFUL PROGRAMMER'S TOOL. A great deal of this power stems directly from the fact that MacApp is written in an object-oriented programming language, and that this tool can easily be enhanced by you—the application programmer. However, to take advantage of MacApp, you must be working in an object-oriented programming language, such as Object Pascal. This chapter is not meant to be, and couldn't be, a complete guide to object-oriented programming. However, if you have a solid programming background, you should be able to learn the essentials of Object Pascal by reading this chapter—even if you have no experience with object-oriented programming. Chapter 3 then fills in the details by describing the exact syntax of Object Pascal. ■

The big picture

Before getting into the specifics of object-oriented programming, this section describes the theory behind this new way of programming, to show exactly how and why it differs from conventional procedural programming in languages such as Pascal or C.

In more conventional programming environments, the organization of a program is centered around the procedures and functions of that program. Typically, a program solves a problem by dividing it into a list of simpler tasks, each divided into simpler tasks still. This gives programs a tree-like structure organized around their routines.

■ **Figure 2-1** Tree-like structure based on routines



This conventional programming paradigm allows large programming problems to be solved by dividing programs into routines. Therefore, problems that can be broken up into tasks that execute in a fairly predictable, linear manner are easily solved. Unfortunately, on a computer like the Macintosh, the user interface presents a programming task that is typically not predictable or linear.

Object-oriented programming allows programmers to solve complex, nonlinear problems in small, easily-managed functional units. This process requires forgetting the old habit of breaking programs into linear algorithmic solutions. Instead, the idea is to break problems up into functional units called **objects**.

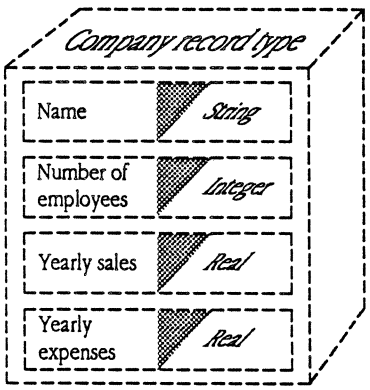
Theoretically, an object is a data structure in memory that provides some function in addition to some data storage. In this way, objects are equivalent to little computers in memory, storing data, doing calculations, and communicating with other objects. Dividing the responsibilities wisely among different objects can lead to well-organized programs that can solve monumental tasks.

Although it is not an entirely accurate representation, it might be easier at first to think of objects as records (in Pascal) or structs (in C) with some new features. In fact, the way objects are declared and the way they use data fields are strikingly similar to record and struct declaration and data field use.

Record types and record variables

If you are familiar with the use of records, then you already have a beginning understanding of objects, because objects are analogous to records with extended functionality. When you decide what types of records you want to use in a program, you create **record type definitions,** which name the record types and declare the data fields belonging to each record type. These definitions don't actually create any records, however. In the same way that a cookie cutter defines the shape of a cookie but not its contents, the record type definition defines the shape of a record variable but contains no data itself.

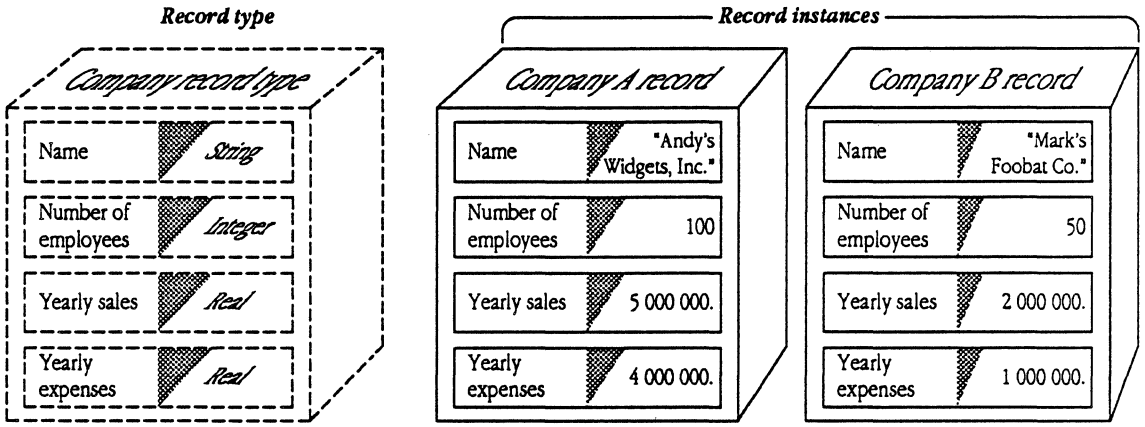
■ **Figure 2-2** A record type



For instance , in order to store information about a business, we might declare a Company record type like the one in Figure 2-2. Once a record type has been defined, actual records of that type can be created when the program is executing. To create actual **instances** of a record type, you would typically declare variables of that type. These variables will represent actual space in memory where data can be stored.

It is important to remember that the record type definition has no space allocated in memory. Only after a record variable of that type is declared is a space in memory reserved for storing values in the data fields. The distinction between the type definition and the actual record in memory is a very important one.

■ **Figure 2-3** A record type and record instances



Classes and Instances

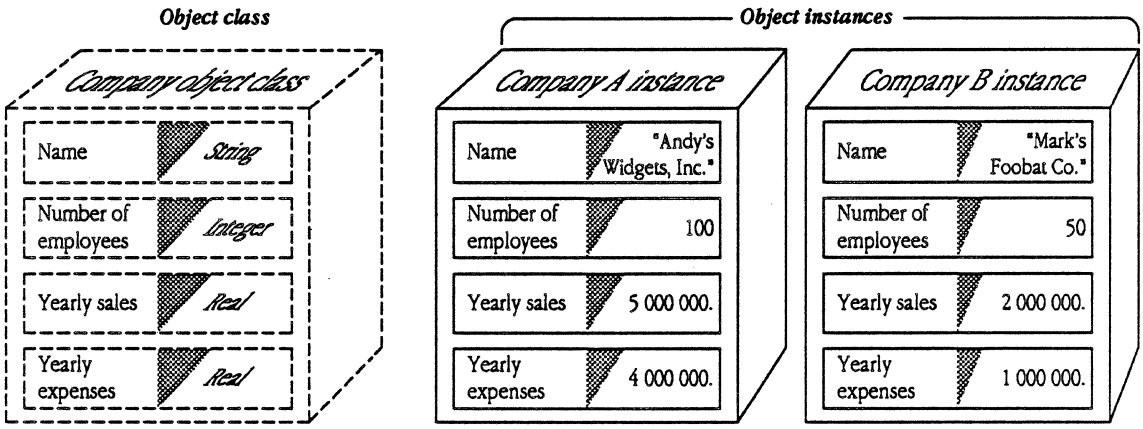
Objects, then, are similar to records with some added features. An object consists of its data and the routines that operate on that data. Like a Pascal record or a C struct, the data that belongs to an object is stored in **fields.** The routines that manipulate the data in those fields are called **methods**.

As with Pascal records, objects use type definitions to define their "shape"—that is, the number and nature of their fields and methods. To help distinguish record types from object types, object types are called **object classes,** or simply **classes.** Keep in mind that type and class mean approximately the same thing.

The actual objects that exist in memory while a program is running are called **object instances**, or simply **objects**. The terms *object* and *instamce* refer to the same thing; another way to say "object" is "an instance of *ClassName*".

Objects are like small pseudocomputers. Each instance has its own memory for data (the fields) and its own functionality (the methods that work on those fields). As a simple example of a class, let's look at a class with data fields but no methods. In Figure 2-4, notice that classes with no methods are conceptually equivalent to record types.

■ **Figure 2-4** A class and two instances with no methods

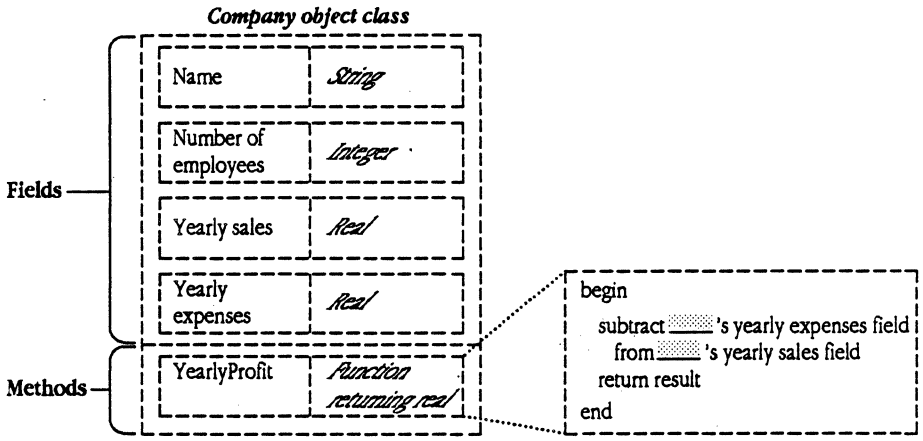


You can only use this Company class to create instances that are very limited pseudocomputers, because any instance of the Company class has storage ability but no functionality. To give functionality to instances of the Company class, you must add a method to the Company class definition.

Let's create an example method for the Company class. Since an instance of the Company class represents an individual company, you might find it useful to have a Company instance be able to calculate its yearly profit by subtracting its yearly expenses from its yearly sales.

Let's call this method YearlyProfit; its definition might look like Figure 2-5.

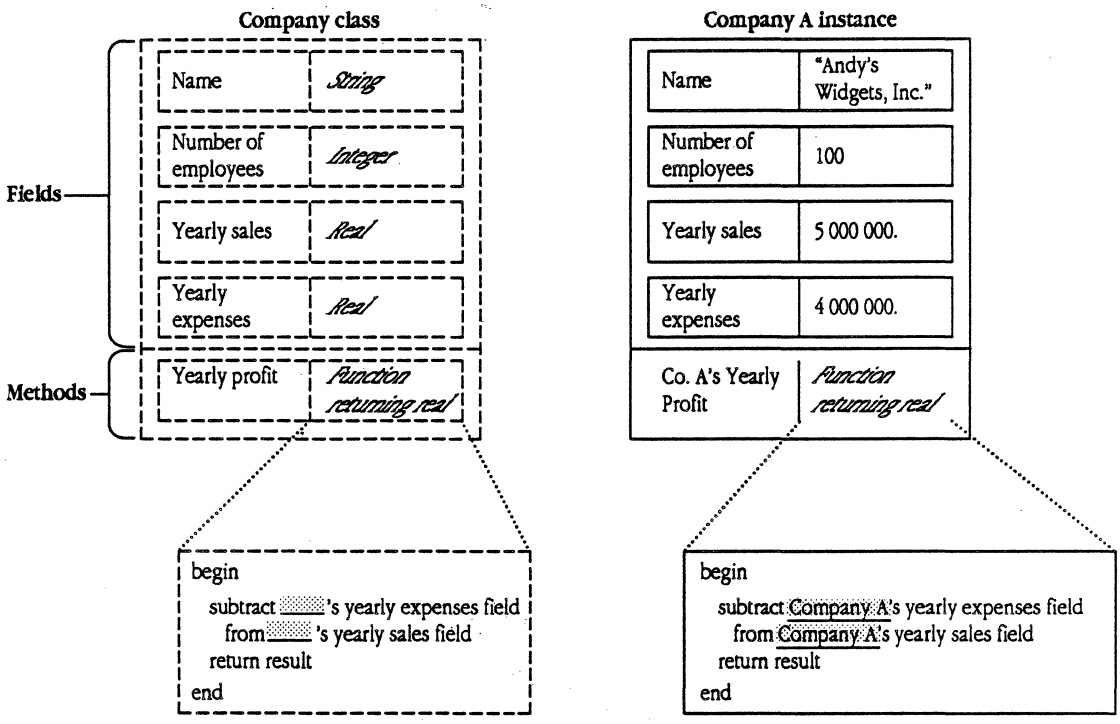
■ **Figure 2-5** A class with a method definition



Because this method belongs to the entire class of Company objects, you cannot simply call it as you would a procedure or function. If you did, how would it know which instance's YearlySales or YearlyExpenses field to use? Remember, a *class* cannot store values in fields—only *instances* can, and there can be many instances of the same class. Therefore, you must call a method to operate on a particular object. To do this, you must **send a message** to that object, telling it to execute the method on itself.

When you send a message to a particular object, the method operates on that particular object's data fields. For example, in Figure 2-6 you can see the differences between the YearlyProfit method as it was *defined* for the Company class, and as it actually *executes* when you send the YearlyProfit message to the Company A object.

■ **Figure 2-6** A method as it is defined and as it executes



The way fields and methods relate to classes and objects is the single most important concept in object-oriented programming. Here is an overview of this relationship:

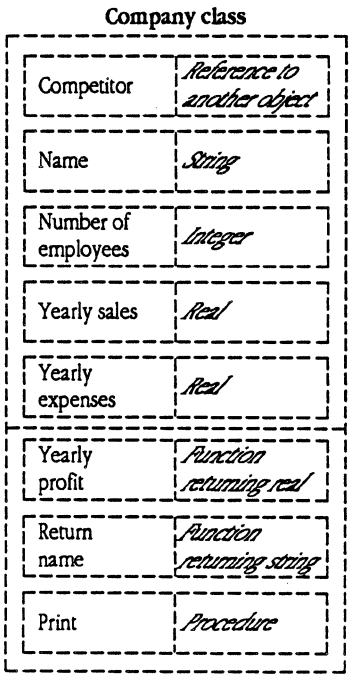
- Fields are defined to belong to an entire class of objects.
- The content of a field belongs to an individual object. When you create an object, space in memory is reserved for that instance to store values for each of the fields of its class. An object can store values in its own fields that are different from the values stored in other instances' fields, as in Figure 2-4.
- Methods are defined to apply to an entire class.
- When you send a message, you send it to a particular instance. When the method executes, it uses the data from the fields belonging to that specified instance.

To summarize, a class definition defines the generic shape of one class of objects—the number and size of the data fields, and the definition of the methods. An object is an actual instantiation of a class—it has its own data in its fields, and you can send it a message telling it to call any of the class's methods on itself.

Flow of control in object-oriented programs

An object-oriented program is one based on objects instead of routines. When an object-oriented program executes, many objects are created. They exist as space in memory where they store their data. In the environments you'll be using with MacApp, there is still a conventional main procedure to start things rolling, but then the instances of the various classes take over. One object's methods will send messages to other objects, as the flow of control bounces around from one object to another. To give a clear picture of how control flows through an object-oriented program, let's first expand the class defined previously.

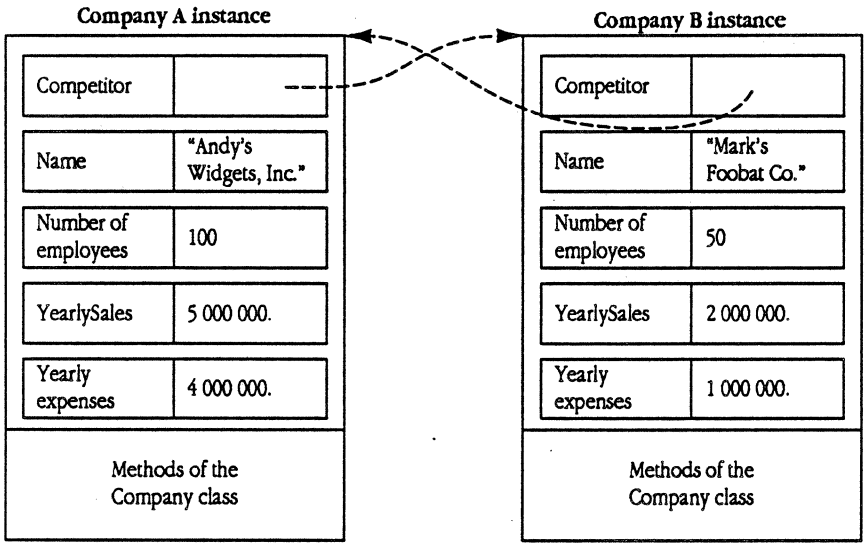
■ **Figure 2-7** An expanded Company class



Notice that this class has a new field—the Competitor field. This field will be used by one object to refer to the object that represents its competitor. In other words, this field is like a pointer that allows one object to reference another object.

For example, in your program you can instantiate this expanded Company class by creating two objects, and making them each other's competitors.

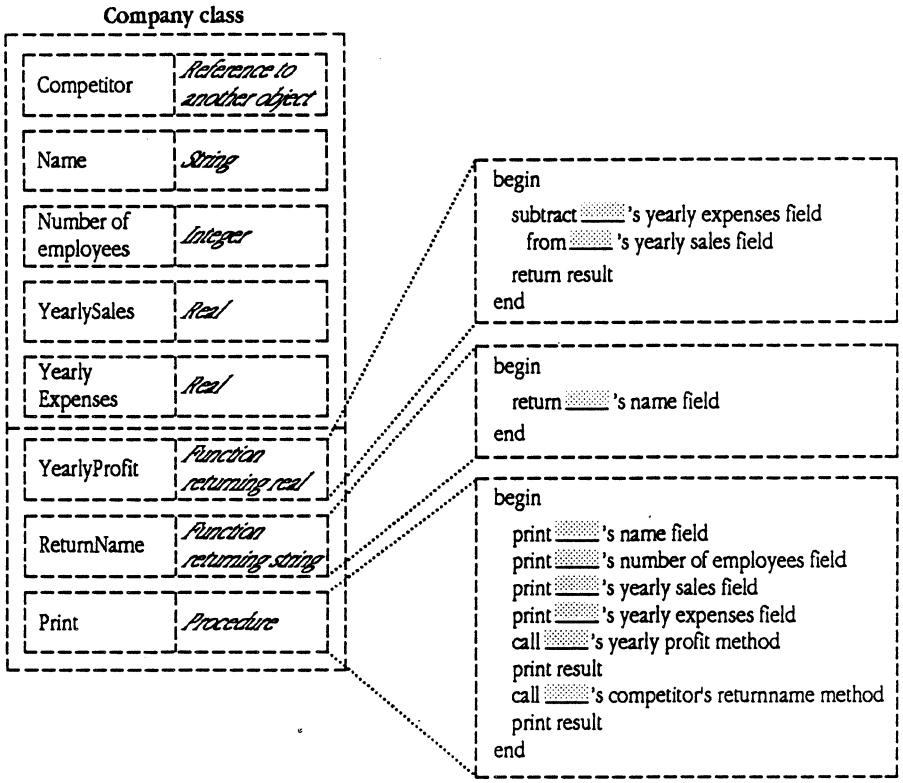
■ **Figure 2-8** Instances referring to each other



You'll soon see why it is important for one object to be able to refer to another.

Going back to the new class definition, you'll see that there are also two new methods—a printing **procedure method** (we'll name it *Print*), and a name-returning **function method** (named *ReturnName*). The purpose of the printing procedure method is to print all the information about a particular object. The name-returning function method will just return the contents of its object's name field to whatever piece of code called it. Figure 29 shows the generic method definition for all three Company class methods.

■ **Figure 2-9** The completely defined Company class



You've seen the YearlyProfit method in Figure 2-5. The ReturnName method is just as simple. It merely returns the contents of the name field as its result. The Print method is also fairly straightforward. Let's examine it in detail, but first let's introduce some terminology.

When you call a method, the following steps occur:

- You must send a message to a particular instance, telling that instance to execute one of its methods.
- That instance then calls the requested method, which was defined for the entire object class, but which will execute specifically on this instance's data.

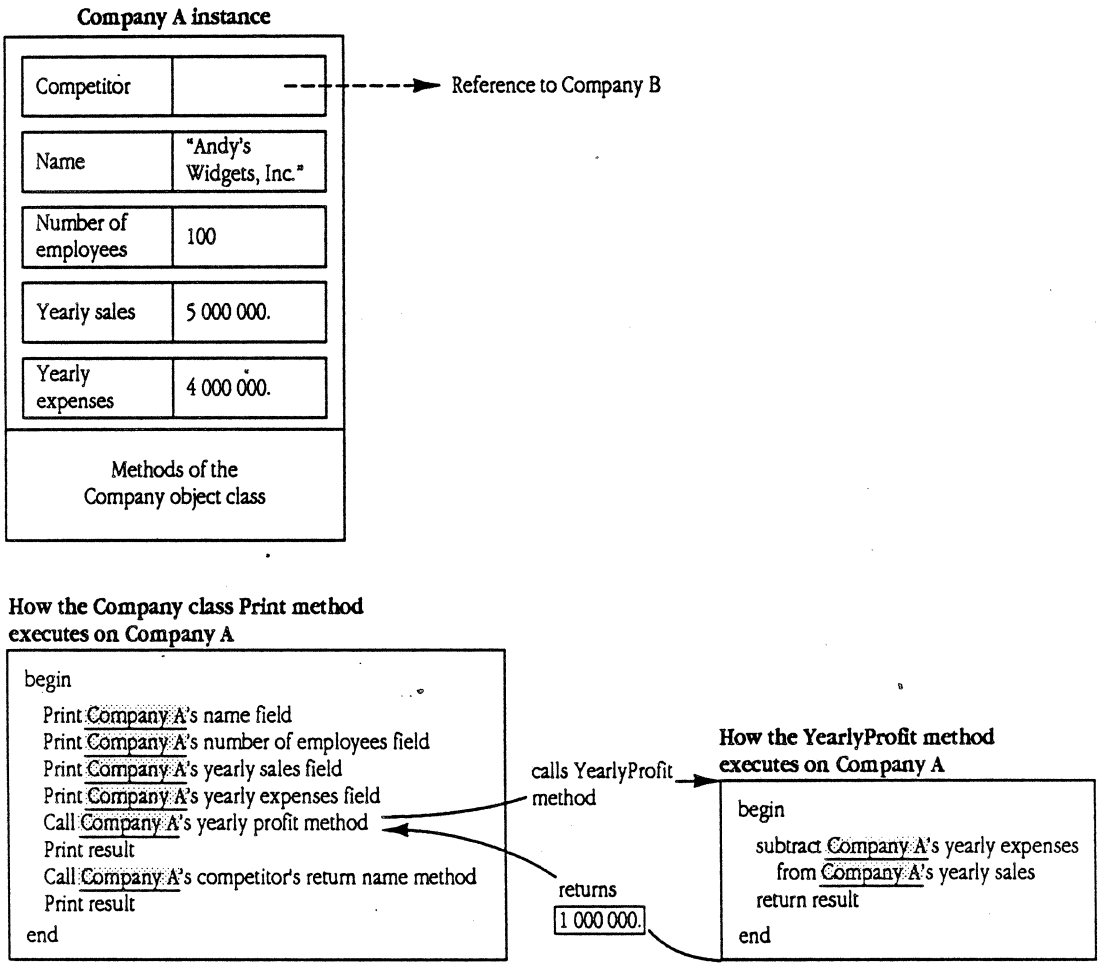
The object that you send the message to, which then calls the requested method, is referred to as the **calling instance**.

The first four lines of the Print method print out the values *of the calling instance's* fields. The next two lines call the YearlyProfit method *of the calling instance*, and then print the result of that method call. The next two lines call the ReturnName method of the instance referred to by the Competitor field and then print the value returned by that method.

Now you can see why it is important to have one object be able to refer to another object: The last piece of information this method prints is actually information about a different object—the *name* of its *competitor.* An object normally does not have access to another object's name fields, but only to its own fields. References between objects allow the methods of one object to send messages to another object.

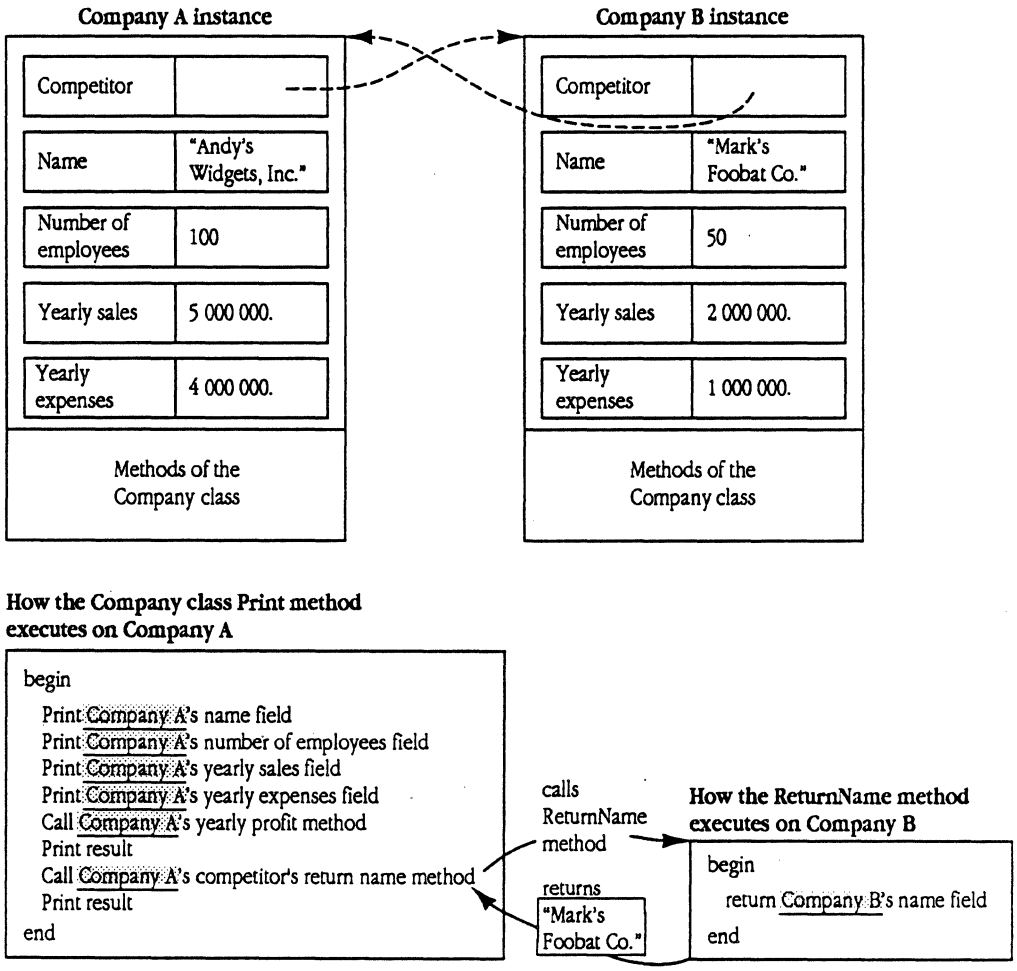
When the printing method of Company A is called, the first four fields of Company A will be printed. Then Company A's Print method will call the YearlyProfit method on itself, as shown in Figure 2-10.

■ **Figure 2-10** A method sending a message to the calling instance



After Company A's Print method prints the yearly profit, it calls the ReturnName method of whichever instance is referred to by Company A's Competitor field, which in this case happens to be Company B.

■ **Figure 2-11** A method sending a message to a different instance



Company B's ReturnName method returns the contents of Company B's Name field (which is "Mark's Foobat Co.") to Company A's Print method, and then A's Print method prints that result. Company A's Print method then ends and returns control to the piece of code that called it. Each of Company A's fields has been printed, A's profit has been printed, and the name of A's competitor has also been printed.

Notice that Company A knows Company B exists because Company A has a reference to Company B in the Competitor field. If Company A's Competitor field pointed to some other Company instance (for example, a Company C), then A's printing method would have called the ReturnName method of Company C.

In other words, the last two lines of Company A's Print method call the ReturnName method of whichever object instance is pointed to by Company A's Competitor field. In this example, it happens to be Company B.

To finish the big picture of object-oriented programming, let's look at the flow of control in a complete, if arbitrarily simple, program written in an English-like programming language.

In MacApp, there is a main procedure just like the ones found in standard Pascal or C. The purpose of this main procedure is straightforward. First, it must allocate space in memory for any **global objects**—instances that are created by the main program. Then the main program must initialize those instances, typically by calling their initialization method if they have one. Finally, the main procedure must call the methods of those instances to get things rolling.

In applications created with MacApp, the main procedure creates one global object, calls its initialization procedure, and then calls its first method. The methods of this global object then do the rest of the work. Other objects may be created, and many methods will be executed, but the main procedure does nothing else itself. In this simplified example, though, the main procedure will create two objects and call one method of each.

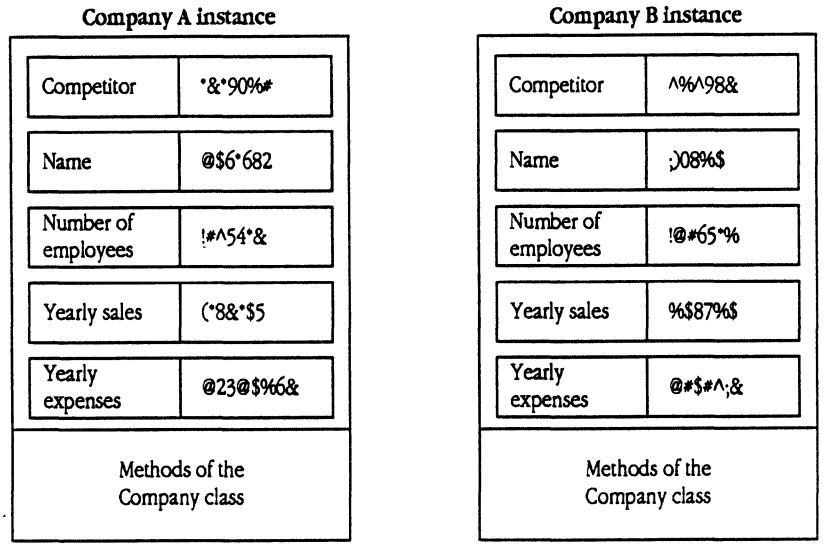
Let's expand the Company class so that it includes an initialization method. You'll see a lot of information on initialization methods a little later, so for now just assume there's a working initialization method.

The main procedure will follow a structure like this:

```
main begin
    create an instance of the company class called Company A
    create an instance of the company class called Company B
    call the initialization method of Company A
    call the initialization method of Company B
    call the printing method of Company A
    call the printing method of Company B
main end
```

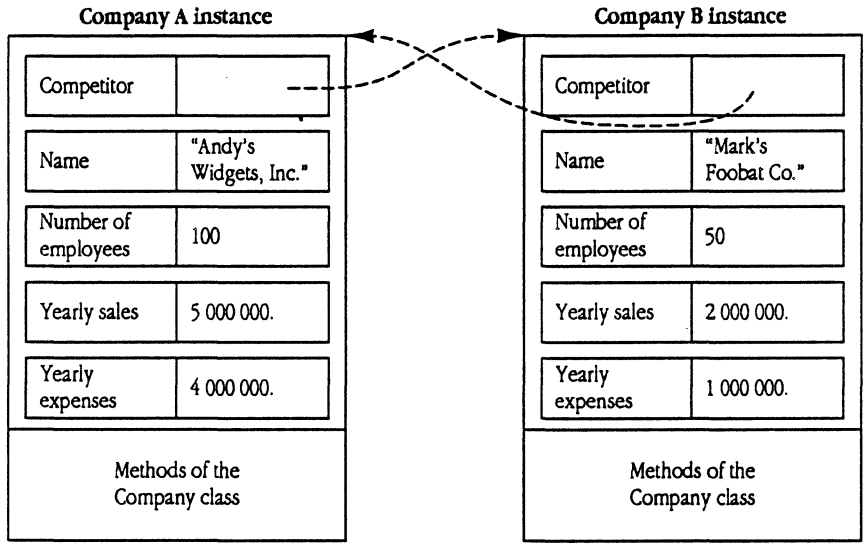
After the first two lines of this main procedure have been executed, the objects will exist in memory, as shown in Figure 2-12.

■ **Figure 2-12** Objects exist in memory



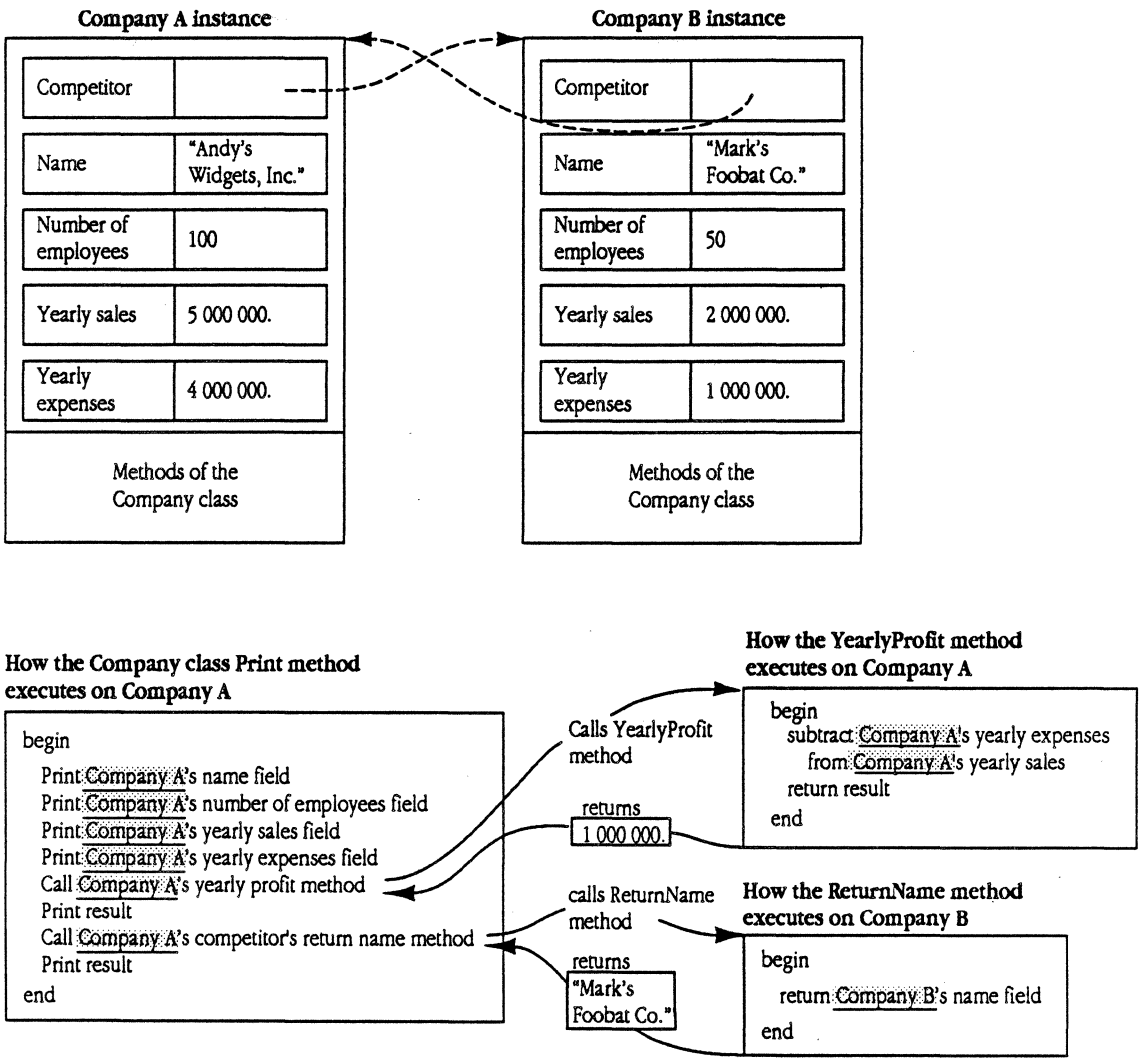
After the initialization methods are called, the objects' fields should all have values. This is true even of the competitor field that links the two objects. (This initialization method will be examined in further detail in later chapters.)

■ **Figure 2-13** Objects have been initialized



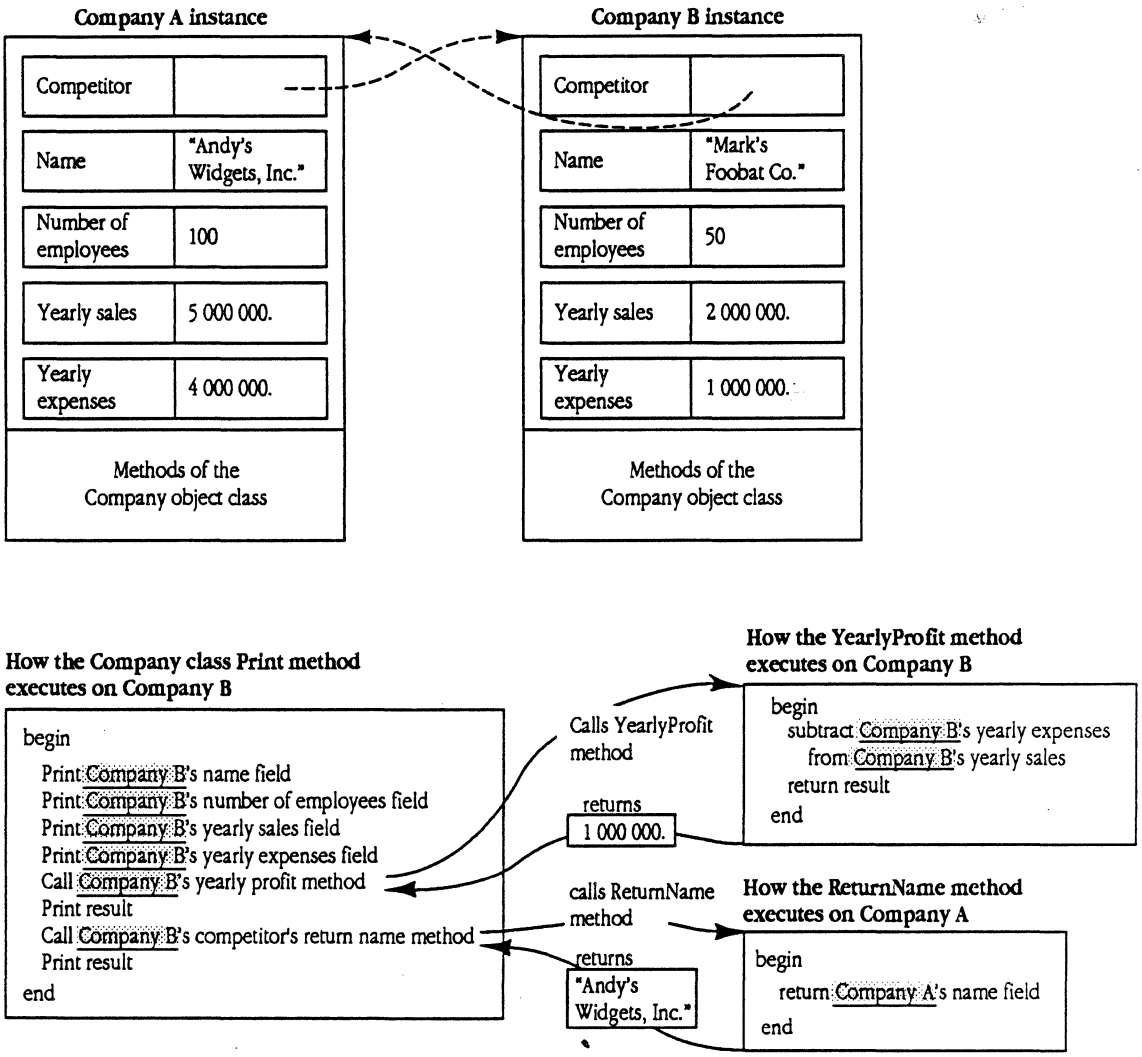
The next step is for the main procedure to call the printing method of Company A. This leads to the flow of control shown in Figure 2-14.

■ **Figure 2-14** Company A's printing method is called



Finally, the main procedure calls Company B's printing method. This flow of control finishes the program:

■ **Figure 2-15** Company B's printing method is called, and the program is finished



The output from this program (given a sophisticated printing routine) should look something like this:

```
The name of this company is: Andy's Widgets, Inc.
The number of employees is:  100
The yearly income is:        $ 5,000,000.00
The yearly expenses are:     $ 4,000,000.00
The yearly profit is:        $ 1,000,000.00
The competitor is:           Mark's Foobat Co.
The name of this company is: Mark's Foobat Co.
The number of employees is:  50
The yearly income is:        $ 2,000,000.00
The yearly expenses are:     $ 1,000,000.00
The yearly profit is:        $ 1,000,000.00
The competitor is:            Andy's Widgets, Inc.
```

In summary, you've seen the difference between records and objects—that objects have methods as well as data fields.

You've also seen that classes, like record types, define the generic shape of object instances by defining both the number and size of the data fields as well as the class's methods.

Finally, you've seen three ways in which an object method can be called:

- The main procedure can send a message to a global object instance.
- A method of an object instance can send messages to other methods of that instance.
- A method of an object instance can send messages to other instances, as long as the first object has an object reference field that refers to the second object.

Classes

Object-oriented programs are organized at two levels. The first, which you've already seen, is the organization that exists during runtime between instances. One instance creates a reference to another instance through object reference fields, and can subsequently make calls to the other instance's methods. This is the flow of control of an object oriented program. The second level of object organization involves class definitions. Unlike record types, each of which is wholly independent of every other type, classes can be organized into hierarchies, where classes inherit characteristics from other classes. You can put a great deal of structure in your program by creating well-organized class hierarchies.

The hierarchical structure of object classes is the basis of an object-oriented program's organization. Object classes are organized into hierarchies where each class can be the **descendant** of another class, called its **ancestor class** or **superclass**. Each class can be a superclass to any number of **descendant classes**, or **subclasses.** Classes are therefore organized into tree structures.

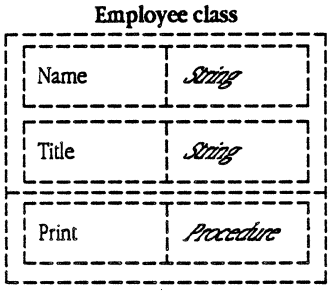
- *Note:* In some object-oriented programming environments an object class can be the descendant of more than one other class. Although this can be a powerful tool, it can also lead to overly complex class organizations. In the MacApp environment, you will be dealing with classes that descend from only one other class, and all class organizations will be tree hierarchies.

A descendant class contains not only its own fields and methods, but also those of its superclass. Of course, that superclass contains the fields and methods of its superclass as well. This concept, whereby an object class gains extra fields and methods from its ancestor classes, is called **inheritance**.

Inheritance

Let's look at an example of class hierarchy by creating an Employee class. An Employee class might have fields for the employee's name and title, as well as a method for printing the information in these fields.

■ **Figure 2-16** An Employee class

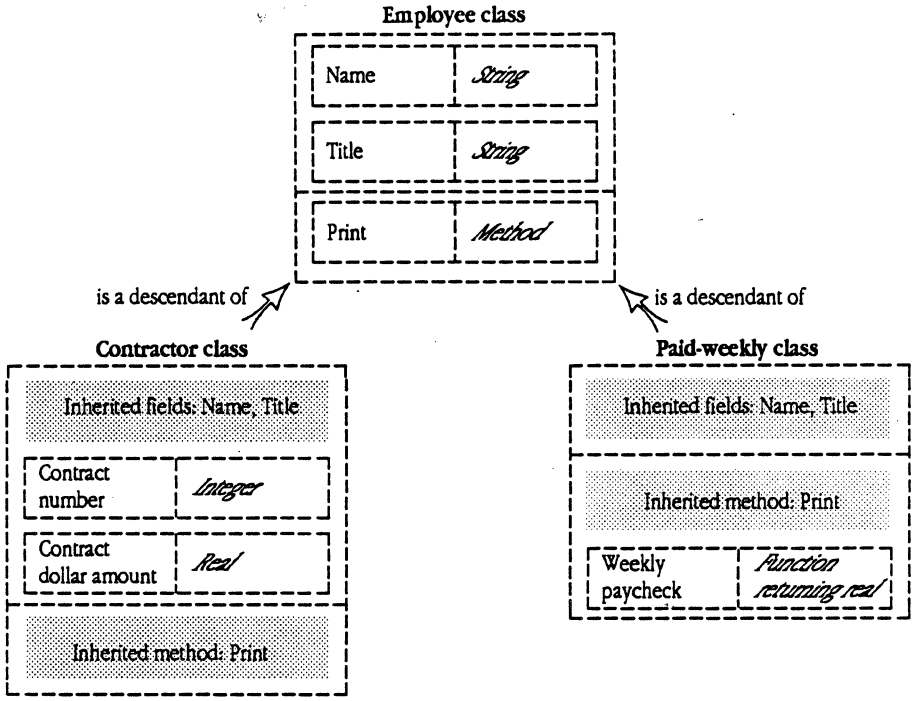


This class, because of its generality, might not include fields and methods for all the important information about each kind of employee that a company might have. There may be many different kinds of employees, and each type of employee could have its own special class. For example, there could be both contractors and paid-weekly employees who have different payroll needs. You could make both of these more specific employee classes subclasses of the more generic Employee class, which would allow you to share fields and methods common to both, such as the ones for storing and printing the employee's name. You could also create customized methods for computing the actual paycheck, which would be different for the two types of employees, and the organization of your object class hierarchy would insure that the proper method was always invoked for the different types of employees.

In an example such as this one, subclasses are **customizations** of their superclass. A Contractor employee class might require two extra fields—a ContractNumber field and a ContractDollarAmount field, for instance. A PaidWeekly employee class might require one extra method—a method for calculating a weekly paycheck. But both of these new descendant classes could make use of the inherited data fields (the Name and Title fields) and Print method of their superclass, the Employee class.

Once these two classes are added, the hierarchy starts to look like a tree.

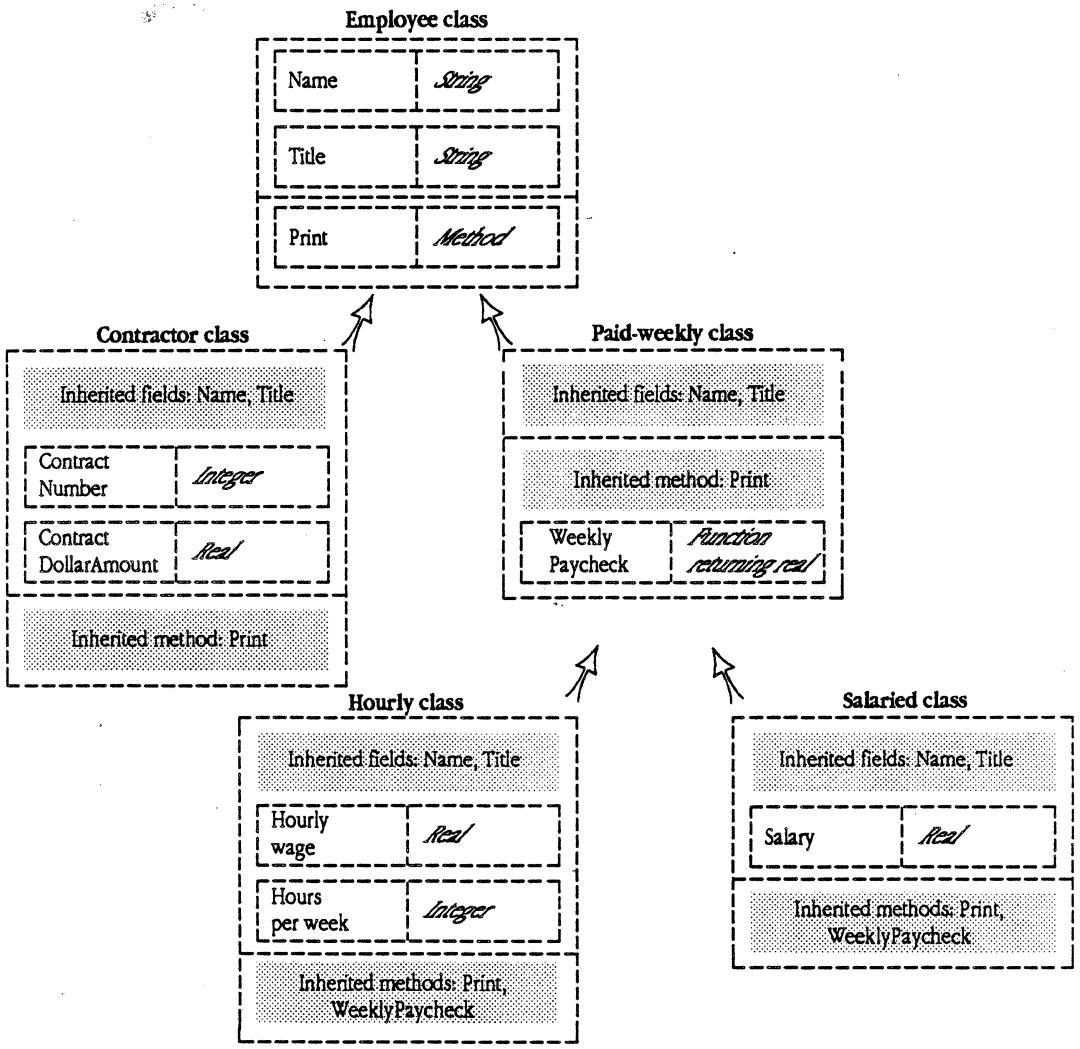
■ **Figure 2-17** A simple hierarchy of classes



In the body of the program, you will be able to create objects belonging to any of these three classes. An instance of the Employee class will have two data fields and one method; an instance of the Contractor class will have four data fields and one method. Two of these fields and one of these methods are inherited from the Employee class; nevertheless, instances of the Contractor class are not able to distinguish between fields and methods that are inherited and those that are not. It's almost as if you declared three separate object classes, but remembered to include the fields and methods of the Employee class in its two descendant classes. Inheritance relieves you of having to remember to do this—subclasses always inherit all of the fields and methods of their superclass.

Inheritance of fields and methods applies through more than one level of ancestry. For example, you can now declare descendant classes of the PaidWeekly employee class. These classes will inherit all the fields and methods of the PaidWeekly class, including those that were already inherited from the Employee class.

■ **Figure 2-18** Multiple levels of inheritance



Once again, objects cannot distinguish between those fields and methods that are inherited and those that are not.

In the above example, the Employee class and the PaidWeekly class are ancestors of the Hourly and Salaried classes. The PaidWeekly class is the **immediate ancestor** of the Hourly and Salaried classes. The Hourly class is a descendant of the Employee and PaidWeekly classes, and is the **immediate descendant** of the PaidWeekly class. While this terminology may be helpful when discussing the relationships that various classes have with one another, level of ancestry has no effect on the functionality of your code. The Hourly class, for example, would still inherit the same fields and methods from the Employee class regardless of the number of classes in the hierarchy between the Hourly and Employee classes.

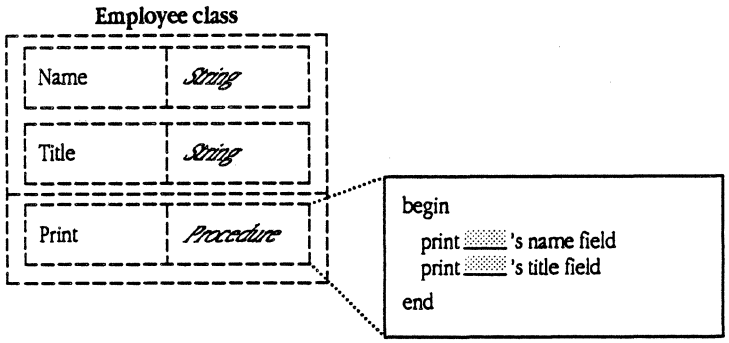
Method definitions

An important part of the class definition is the method definition. Remember that when a method is defined, it belongs to an entire class of objects. Therefore, the definition must be in a general form. However, a method operates only on the data of the object that called it.

Since the method definition states the *generic* method that applies to an entire class of objects, it cannot refer to the fields or methods of any particular object. Instead, the method definition must refer to data fields and methods by their generic class name. When you send a message, you must specify which instance the method is to operate on (referred to as the **calling instance.**) As the method executes, references to the generic field and method names in the method definition will be automatically replaced by references to the specific fields and methods belonging to the calling instance.

As an example, let's define the generic form of the Print method for the Employee class. This method is supposed to print the values of the calling instance's fields. Figure 2-19 shows the generic method definition.

■ **Figure 2-19** The generic method definition of the Print method of the Employee class



Since this is a **method definition**, it applies generically to an entire class of objects. Whenever this method is called, however, it is called to execute on the specific data of some particular instance—the calling instance.

Notice that this method definition references both the Name field and the Title field of the Employee class. Of course, neither of these fields has any value until an instance of the Employee class is created. Once an Employee *instance* is created, you can call its Print method, and the Print method for the Employee class will be executed *specifically on the data of that instance.*

Override methods

A class inherits the methods of its superclass. Usually this is a good thing; that's why you make one class a subclass of another class. Remember, though, that a subclass is a *customization* of a superclass. Therefore it is sometimes desirable to have a subclass that inherits the methods of its superclass, but somehow customizes those methods. Object-oriented programming allows that flexibility.

There are two ways a subclass can change an inherited method. It can either have its own completely rewritten version of the method, or it can add something to the inherited method. Either way, the new method is called an **override** method. The ancestor's method definition is **overridden,** and this process is called **overriding** a method.

Sometimes the override method needs to be so different that it cannot even use the inherited method. In fact, it is quite common for a method to do absolutely nothing, requiring that every class that inherits that method completely override it.

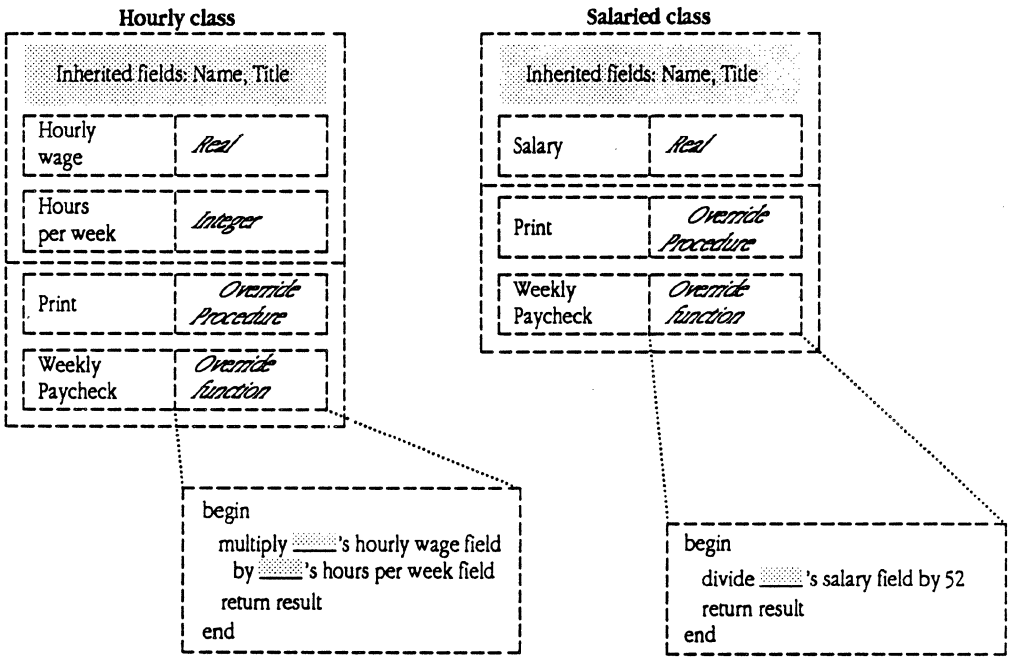
The reasoning behind this is very simple. Consider the PaidWeekly class from the earlier example. This class has a WeeklyPaycheck method but it has no fields that could possibly help in calculating the size of a weekly paycheck; however, the descendant classes of the PaidWeekly employee class (Hourly employees and Salaried employees) do have fields that allow calculation of a weekly paycheck.

The program will not have any instances of PaidWeekly Employee class; this class was created solely to help organize the class hierarchy and therefore is called an **abstract class.** Abstract classes are quite common in object-oriented programming—they provide a common framework of fields and methods for their descendants. Frequently, you will have to override the methods of abstract classes.

Notice that superclasses are not synonymous with abstract classes: Later you will create instances of the Employee class, at the very top of the class hierarchy. Of course, the compiler will allow you to create instances of any class. It's your decision which classes are to be abstract and which are not.

In the case of the WeeklyPaycheck method, you simply leave the definition of the inherited version blank, and write the override methods from scratch.

- **Figure 2-20** An override method that does not call its inherited method



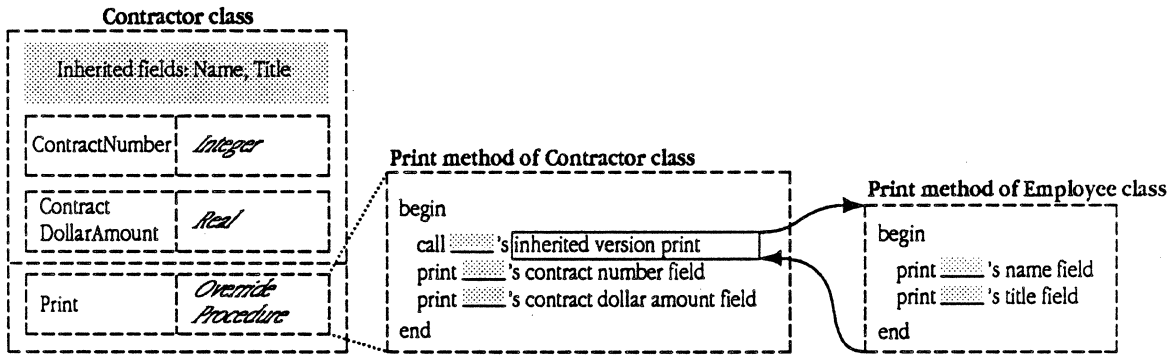
Sometimes the override method does not need to completely replace the inherited method, but only augment it. For example, look at the Print method defined in Figure 2–19. This method prints only the values of the Name and Title fields. This is fine for instances of the Employee class; however, this method is inherited by both the Contractor and the PaidWeekly classes. In the case of the PaidWeekly class, the inherited method will be fine because the PaidWeekly class doesn't have any new fields. However, the Contractor class has two fields that the Employee class does not: the contract number and the contract dollar amount. The Contractor class, then, needs a Print method that augments the Print method of the Employee class.

This override method ought to:

- call the inherited method, which will print the Name and Title fields
- print out the ContractNumber and ContractDollarAmount fields itself

This is exactly what you can define the override method to do.

- **Figure 2-21** An override method that calls the inherited method



Now that the class hierarchy is defined, it is possible to create instances and examine how inherited and overridden methods work during runtime.

Objects

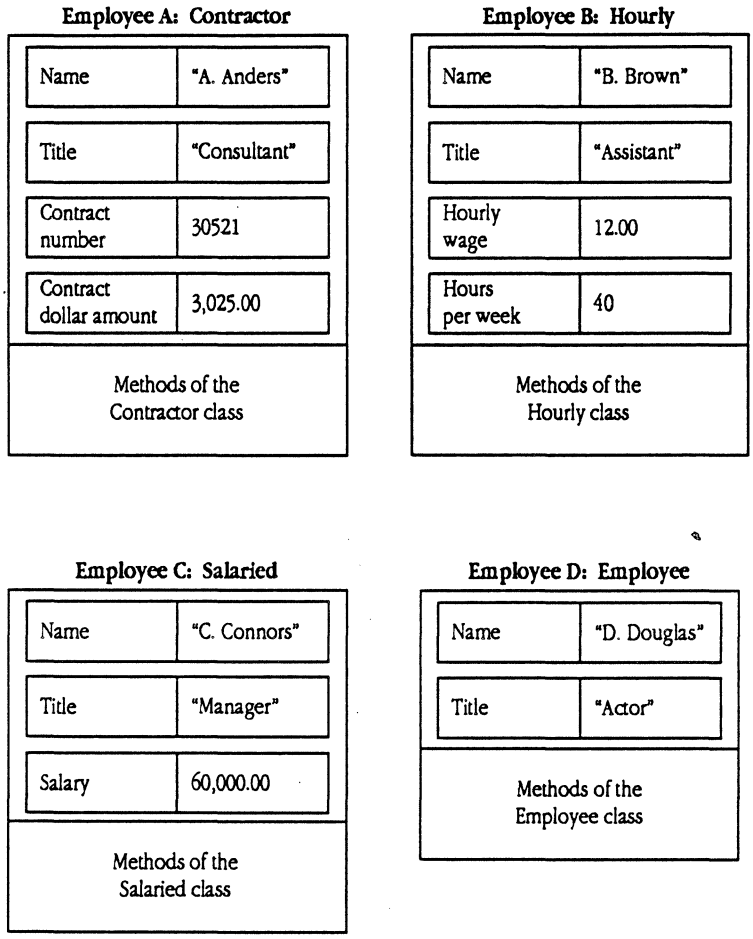
To create a class, you declare the methods and fields of that class and define the methods. When you create an instance of that class, the compiler will allocate space in memory for the fields of that instance and set up a mechanism for the instance to call its methods when it receives messages.

An instance method

To begin, let's create an instance of each employee class defined except for the PaidWeekly class, which is an abstract class used only for hierarchical organization. For example, you might want to create four Employee instances:

- Employee A, a contractor (an instance of class Contractor)
- Employee B, a paid-hourly employee (an instance of class Hourly)
- Employee C, a salaried employee (an instance of class Salaried)
- Employee D, an employee that doesn't fit any of the above categories (an instance of the most generic class—Employee)

- **Figure 2-22** The four Employee instances



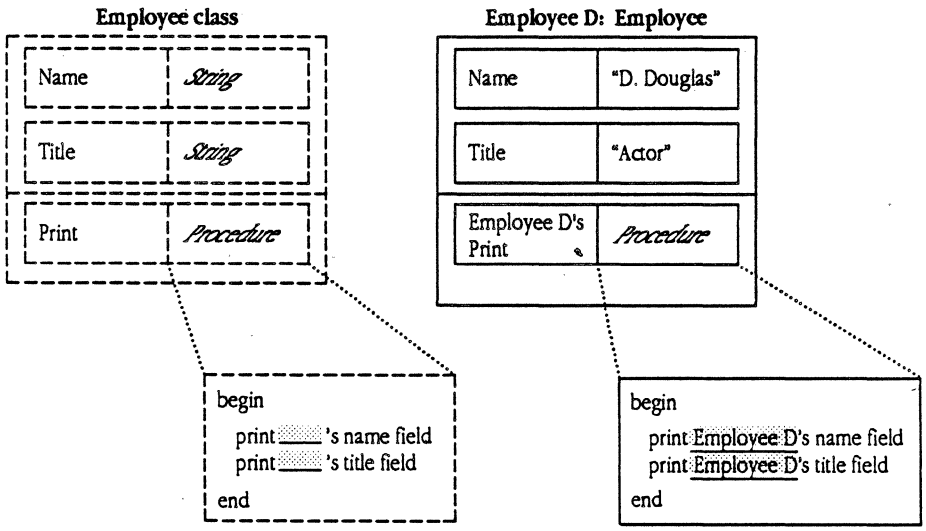
An object instance is said to be a **member** of its class and of all its ancestor classes. Therefore Employee A is a member of both the Contractor class and the Employee class. Employee B is a member of the Hourly class, the PaidWeekly class, and the Employee class. Employee C is a member of the Salaried class, the PaidWeekly class, and the Employee class. Employee D is a member of only the Employee class.

Perhaps you might think that because the Contractor class is a subclass of the Employee class that Employee A is a descendant of Employee D. In fact, the Employee A object has no relationship to Employee D, since these are objects and only *classes* have that sort of hierarchy. *Instances* can only reference each other through the use of **object reference fields,** as in our previous example where each of our Company instances referenced the other's Name field by using their own Competitor field.

These objects can have three types of methods: methods specific to their class, methods inherited from a superclass, and inherited methods that are overridden.

Employee D, which is of the Employee class, has a printing method defined by the Employee class. When the Print method of instance Employee D is called (by the main program, for example), the Print method references and prints the data fields of Employee D, then returns control to the main program.

- **Figure 2-23** The Print method definition and a specific object



The case with methods that are inherited from an superclass is exactly the same. When an inherited method of an object is called, the method executes specifically on that object's data. For example, if you were to create an instance (let's say Employee E) of the PaidWeekly employee class and call the Print method (which is inherited from the Employee object class and not overridden), the method that would execute would look the same for Employee E as the Print method in Figure 2-23 looks for Employee D.

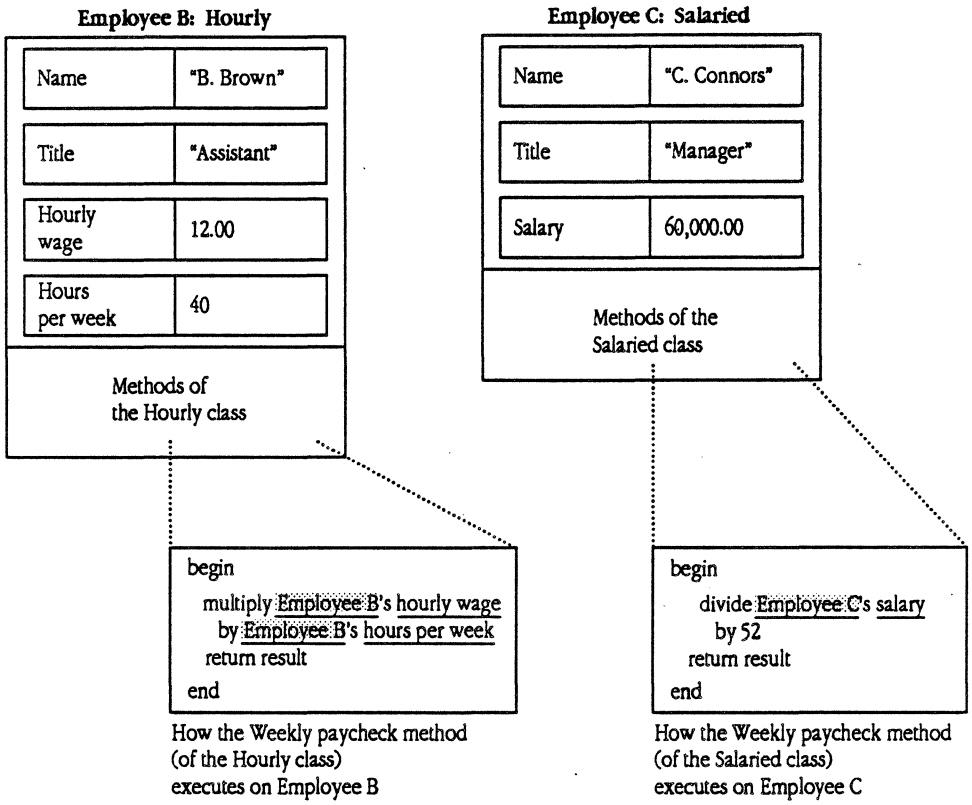
Inheritance and override methods

Override methods that don't call their inherited method

This category is the simpler of the two. If an override method completely overrides the inherited method, then it acts just like an entirely new method. For example, Hourly employees and Salaried employees both have a WeeklyPaycheck method that completely overrides the WeeklyPaycheck method of the PaidWeekly employee class, as shown in Figure 2-20.

Figure 2-24 shows how a call to Employee B's WeeklyPaycheck method is quite a different thing from a call to Employee C's WeeklyPaycheck method. First of all, the methods actually execute on the data of the calling instance. Second, object-oriented programming makes sure that the correct version of the method is executed. For Employee B, the correct version of the WeeklyPaycheck method is the Hourly class version. For Employee C, the correct version is the Salaried class version.

■ **Figure 2-24** The WeeklyPaycheck methods of the Hourly and Salaried classes



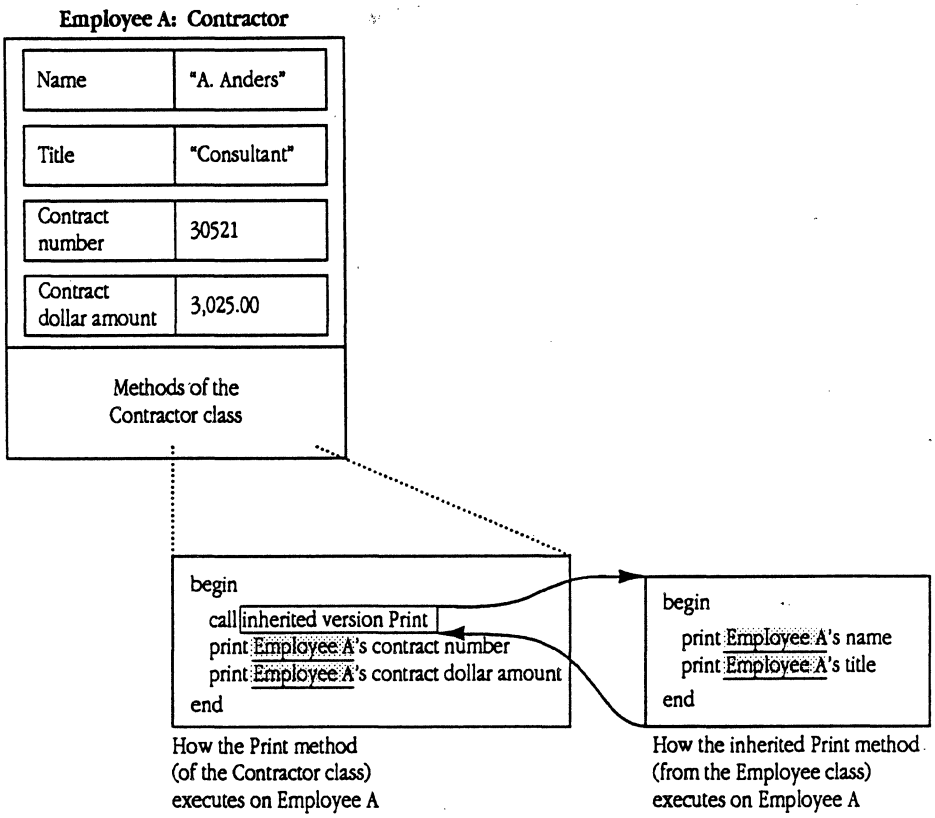
As before, when either of these methods is called, it executes linearly and then returns the result to the calling line of code.

Override methods that do call their inherited method

As we have noted earlier, subclasses may often be used to further customize fields or methods inherited from their superclasses. Inheritance in object-oriented programming allows you to accomplish this easily by creating an override method that calls the inherited method and then uses the result returned to it to complete its task. Consider the example of the Print method of the Contractor class, as shown in Figure 2-25.

When the Print method of Employee A is called, it first calls the inherited Print method, which prints the Name and Title fields of Employee A. Then the override Print method itself prints the ContractNumber and ContractDollarAmount fields of Employee A. Finally, the method returns control to its caller.

■ **Figure 2-25** A method that calls an inherited method



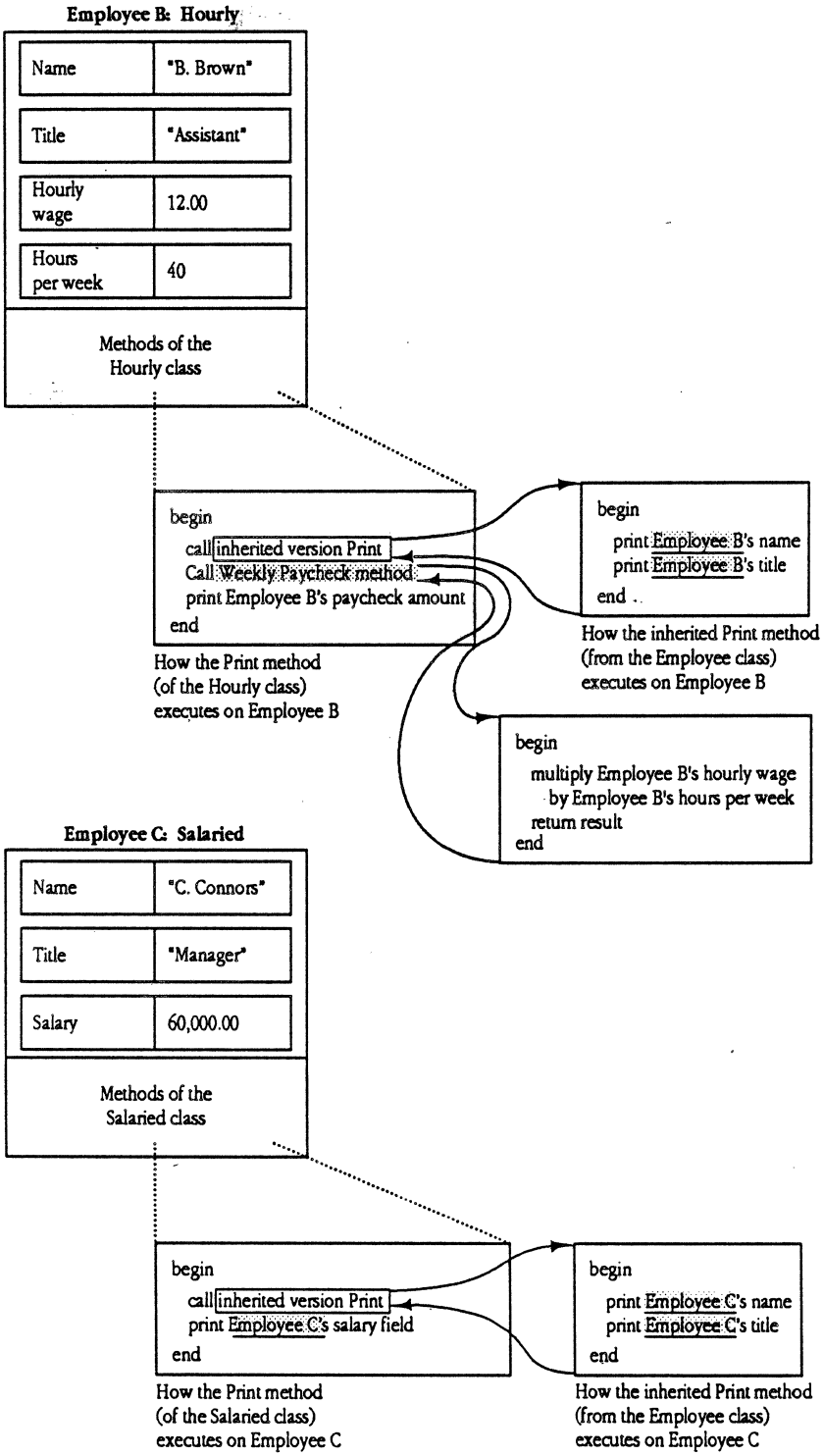
Take notice of these three points:

- Employee A's Print method is called; therefore the printing method that executes operates specifically on Employee A's fields.
- Employee A's Print method makes a call to the Print method inherited from the ancestor object class—the Employee class.
- This inherited Print method also operates only on Employee A's fields.

As another example, consider the Print methods of the Hourly and Salaried classes. These are both override methods, overriding the Print method of the PaidWeekly class, which is inherited from the Employee class.

When the Print methods of Employee B and Employee C are called, the methods that execute would look like those in Figure 2-26.

■ **Figure 2-26** More examples of override methods called by specific instances



Privacy between instances

The Employee objects discussed in this section do not have any fields that point to other objects, as the Company objects did in the first section of this chapter. This means that no Employee object knows of the existence of any other Employee object. Because of this, no method of Employee A, for example, can make reference to any field or call any method of Employee B.

In the Company object example, however, Company A had a link to another object, Company B, and vice versa. In that example, the methods of one object could reference the fields or call the methods of another object.

In Object Pascal, one object can access any field or method of another object, as long as the first object has a reference to the second object. In general, the data stored in an object's fields should be considered private, which means that an object can examine and manipulate its own data (in its methods) but should not examine or manipulate the data in another object. Therefore, your method definitions should only directly examine or manipulate the data fields of the calling instance.

When you adhere to the convention of only allowing methods of an object's class or its subclasses to access its data fields, you significantly reduce your risk of making unwanted changes to other data structures that are dependent on the fields you are accessing. (This statement assumes that your methods accept appropriate parameters and return appropriate results in the first place.) The concept of using the methods of one class to isolate data from methods of a different class is called **data encapsulation** because it in effect surrounds, or encapsulates, an object's data with its own methods that access the data correctly. The idea of data encapsulation is one of the basic concepts behind the structure of object-oriented programming languages, and you should never defeat it without exercising extreme caution.

If one object needs some information from another object, the first can use an object reference field to send a message to the other object asking it to return the value of the desired field. This is how the problem was handled in the Company example. When Company A wanted to print its competitor's name, it called the ReturnName method of the object referenced by Company A's competitor field, Company B. This allowed Company B to access its own Name field, and return the value as a function result, but Company A didn't look into Company B's Name data field directly.

Ramifications of object-oriented programming

Object-oriented programming lets you organize your programs into object classes and subclasses, and inherit fields and methods, which allows for easy expansion of programs. This ease of expansion allows MacApp to implement a lot of your program for you, freeing you to create the specific objects that make your program unique. Finally, this generic application framework doesn't box you into anything, as object-oriented programming allows you to override any part of the framework easily, without even deleting a line of code.

Ease of expansion

In order to add functionality to an application written in a procedural language like standard Pascal or C, you typically must modify code all over your program. In an object-oriented programming environment, you can add functionality by simply creating a new class or modifying an existing one. All of the subclasses will inherit the new methods and fields, as opposed to your having to find every place that would use your new code and rewrite it appropriately. Reducing the number of places you must change your code by hand also reduces your risk of introducing new errors to already-stable code. And the power of inheritance is also what makes it so much easier to implement standard Macintosh application features by using MacApp as a starting point for writing your own application.

Implementing standard features with MacApp

Most Macintosh programs share many similarities because they follow Apple's user interface guidelines for the Macintosh. Since Macintosh applications have a great number of similarities, the engineers at Apple have used object-oriented programming techniques to put together an extremely powerful but completely generic program for the Macintosh. That program is MacApp. MacApp implements all the standard user interface features or creates a framework for you to implement them yourself. It is your job as a Macintosh programmer to create the objects not included in MacApp that do what you want your program to do.

Customization without tears

You might be thinking that MacApp will coerce you into doing things in a way you don't want to, but that's not true. Although MacApp does most things correctly all by itself, sometimes you'll want something done in a different way. In such cases, you only need override the methods of MacApp that you don't want. In general, the more standard your application's user interface is, the more of MacApp you can use unaltered; and the more unusual your application is, the more of MacApp you will have to override.

Chapter 3 **Object Pascal**

NOW THAT YOU'VE SEEN THE ADVANTAGES OF OBJECT-ORIENTED PROGRAMMING, you must be eager to learn the syntax of a real object-oriented language so you can get to work. This chapter specifies the syntax of Object Pascal, the language in which MacApp is written. ■

Object Pascal file organization

The format of an Object Pascal program is similar to the format of a conventional Pascal program. The program must have a header, followed by the constant declarations, type definitions, and variable declarations. These are followed by procedure and function definitions, and, finally, the main procedure.

Of course, there are new elements in an object-oriented program, namely class definitions and method definitions. The standard format still applies, however.

```
PROGRAM ProgramName;
CONST <constant declarations>
TYPE  <type and class declarations>
VAR   <global variables>
<procedure, function, and method definitions>
BEGIN
    <main program code>
END.
```

Since Macintosh programs are typically complex, this linear organization leads to unmanageably large files. The solution is to break the code into two files, using Pascal **units**, which are discussed fully in the *MPW Pascal 3.0 Reference*. As a quick reminder, a Pascal unit is a separate file that has two parts: an **interface** and an **implementation.** The declarations and definitions in the interface part are available to any file that **uses** the unit. The implementation part contains implementation information private to the unit—for example, procedure and function definitions.

The convention for MacApp programs is to break them into two parts: a unit that defines the object classes and methods, and a main program that uses the unit.

The main file, which stores the global declarations and the main program, is organized like any Pascal program. It makes reference to the unit where the object-related code is stored, by means of the USES statement.

```
PROGRAM ProgramName;
USES ObjectUnitName;
CONST <constant declarations>
TYPE <type definitions>
VAR  <global variables>
<procedure, function, and method definitions>
BEGIN
    <main program code>
END.
```

The unit file defines the classes and methods that will be used by the program, and is typically organized like this:

```
UNIT ObjectUnitName;
INTERFACE
CONST <public constant declarations>
TYPE  <class definitions>

IMPLEMENTATION
CONST <private constant declarations>
VAR   <private variables global to the unit>
<method definitions>
END.
```

As this unit becomes more unwieldy itself, it is typically divided into two files. The implementation part is usually removed to another file, which is **included** in the first one. When this is done, the interface file has the following form:

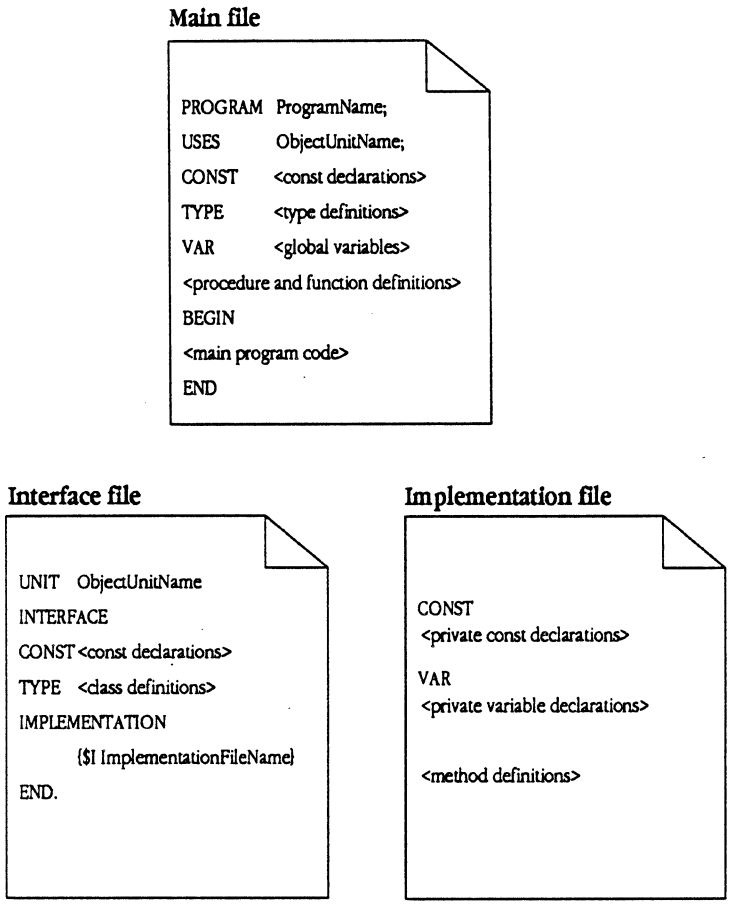
```
UNIT ObjectUnitName;
INTERFACE
CONST <public constant declarations>
TYPE  <class definitions>

IMPLEMENTATION
{$I ImplementationFileName}
END.
```

The implementation file, then, looks like this:

```
CONST <private constant declarations>
VAR   <private variables global to the unit>
<method definitions>
```

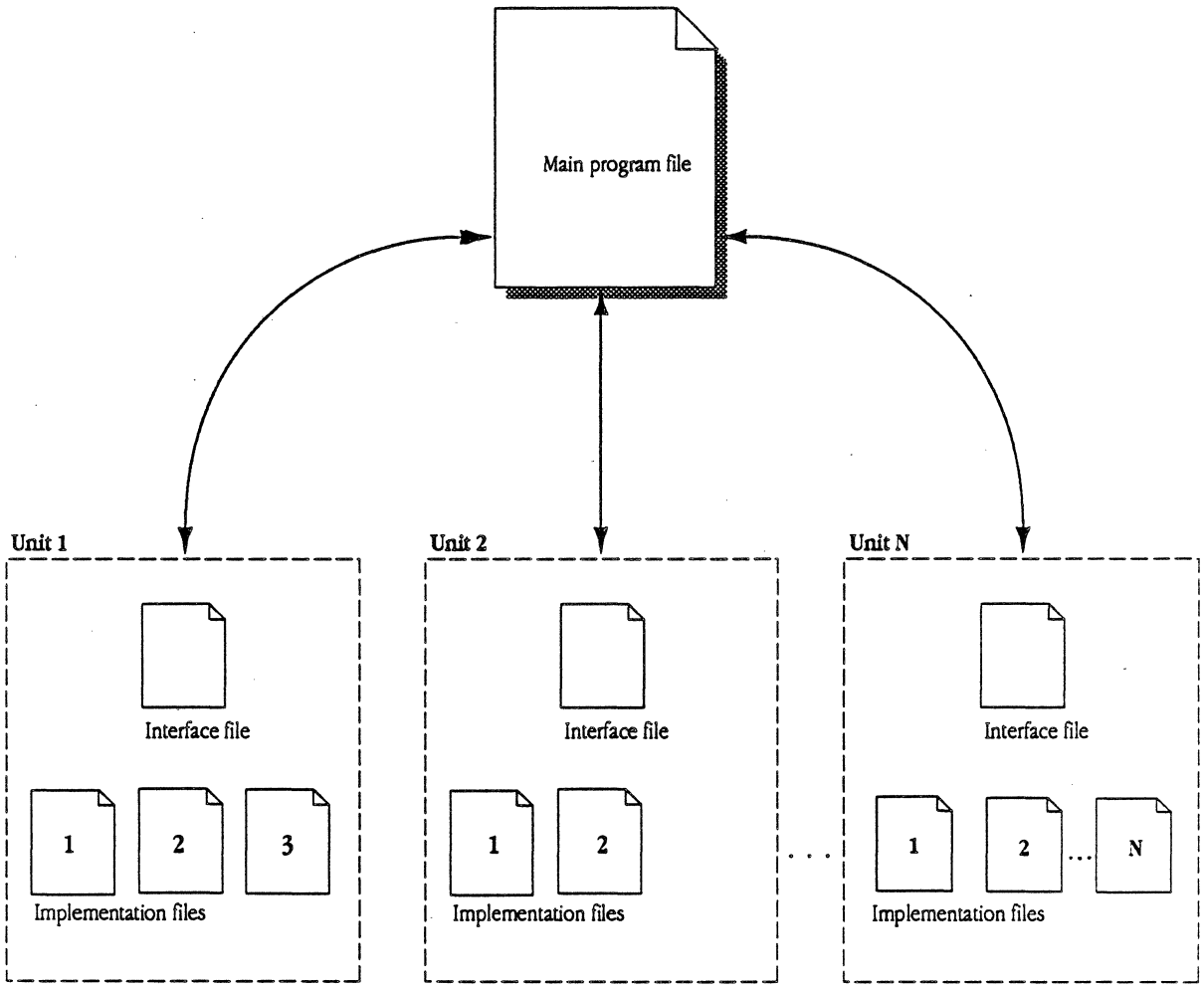
These three files—the main file, the unit interface file, and the unit implementation file—constitute the entire object program, as illustrated in Figure 3-1. You'll see this pattern of files repeatedly when working with MacApp.

■ **Figure 3-1** Typical file organization of an Object Pascal program



As mentioned earlier, you are not limited to three files for your code. You may implement any number of units and any number of implementation files for each unit. In this manner you can keep your files to a reasonable size—and remember, units can be compiled separately. Figure 3-2 shows a possible organization of a large Object Pascal program.

■ **Figure 3-2** Possible file organization of a large Object Pascal program



Class definitions

Let's start to fill in some of the blanks in those file specifications. In the interface part of the unit, there's a section called TYPE where classes and their methods are defined. Defining a class in Object Pascal is similar to defining a record type. First, you need to name the class, specify that it is an object class, and then declare the fields and methods. The specific syntax looks like this:

```
TYPE
      ObjectClassName = OBJECT
            {field declarations}
            {method declarations}
      END;
```

OBJECT is a reserved word that acts almost exactly like the reserved word RECORD. As you can see, the object class definition above has the same structure as a record type definition.

It is the convention in MacApp to begin class names with a *T*. (This stands for Type—remember "class" means "object type.") For example, if you wanted to implement the Company class from Chapter 2, you might make the following declaration:

```
TCompany = OBJECT
      {TCompany's field declarations}
      {TCompany's method declarations}
END;
```

In this declaration, class TCompany has no ancestor. By convention in MacApp, if a class has no superclass, then it is made a descendant of class TObject. TObject is a class with no fields and only a few "utility" methods that will come in handy later. For now, just remember that classes with no immediate ancestor should use TObject as their immediate ancestor.

In Object Pascal, the way to declare the superclass of a class is simple. Since the TCompany class should descend from the TObject class, you can simply rewrite the TCompany definition like this:

```
TCompany = OBJECT(TObject)
      {TCompany's field declarations}
      {TCompany's method declarations}
END;
```

Notice the TObject class name appears in parentheses after the keyword OBJECT. This defines the class TCompany as an immediate descendant of the TObject class.

Field declarations

Now that the class has a name and a superclass, you need to declare the class's fields. By convention, classes are named with an uppercase T and object fields are named with a small f. There's actually a good reason for this, as you will see in the next section. Object fields are declared in precisely the same manner as record fields. For example,

```
TCompany = OBJECT(TObject)
      fMainCompetitor:  TCompany;
      fName:  Str255;
      fNumberOfEmployees:  integer;
      fYearlySales:  real;
      fYearlyExpenditures:  real;
      {TCompany's method declarations}
END;
```

TCompany has five fields. The first field, fMainCompetitor, is a reference to another company object. You'll see more on object references later in this chapter, but for now, remember that an object reference is similar to a pointer to an object. An object reference field, such as fMainCompetitor, allows one object to call the methods of another object. As you can see in the example just given, object reference fields are declared to be of the class that the field refers to. In this case, fMainCompetitor is declared to be of class TCompany, meaning that fMainCompetitor will be used to refer to instances of class TCompany.

The other four fields are straightforward: they exactly mimic the declaration of record fields.

Method declarations

The class definition includes field declarations and method declarations. Method *declarations* should not be confused with method *definitions*. A **method declaration** simply states the name of the method, its parameter list, whether it is a procedure or function, and whether it is an override method. The **method definition** actually defines the method—it contains the code that describes the method's behavior. Method declarations are a lot like forward declarations in Pascal. They describe the method's nature, but do not define the method's behavior.

The TCompany class has three methods. Let's look at the method declaration line for each of them.

```
FUNCTION TCompany.YearlyProfit:  real;
FUNCTION TCompany.ReturnName:  Str255;
```

```
PROCEDURE TCompany.Print;
```

The first two of these methods are function methods. Therefore, their method declaration line begins with the keyword FUNCTION, just as any Pascal function would. The third method is a procedure method, and, similarly, it is declared with the keyword PROCEDURE.

Following the keyword on the method declaration line is the name of the method. Since methods belong to an entire class, the complete name of the method is the name of the class, followed by a period, followed by the name of the method. Always writing the complete method name saves confusion when two or more classes have a method of the same name.

Finally, the function methods end the method declaration line with the specification of the type that is returned by the function. TCompany.YearlyProfit returns a real value and TCompany.ReturnName returns a string value.

Methods, like any procedures and functions, can take parameters. If a method takes parameters, the parameter list comes after the method name, and looks like any parameter definition list. For example, if the three methods above took parameters, their declaration lines might look like this:

```
FUNCTION TCompany.YearlyProfit(LessTaxes: real):  real;
FUNCTION TCompany.ReturnName(Capitalized: boolean):  Str255;
PROCEDURE TCompany.Print(LongForm: boolean; HowManyTimes: integer);
```

Reverting to the earlier method declaration lines, you can now put together a complete class definition:

```
TCompany = OBJECT(TObject)
      fMainCompetitor:  TCompany;
      fName:  Str255;
      fNumberOfEmployees:  integer;
      fYearlySales:  real;
      fYearlyExpenditures:  real;
      FUNCTION TCompany.YearlyProfit:  real;
      FUNCTION TCompany.ReturnName:  Str255;
      PROCEDURE TCompany.Print;
END;
```

Override method declarations

The only aspect of defining classes left to explore is that of declaring certain methods to be override methods of ancestors' methods. For instance, if you wanted to define the employee class from Chapter 2, you might end up with a definition like this one:

```
TEmployee = OBJECT(TObject)
      fName:  Str255;
      fTitle:  Str255;
      PROCEDURE TEmployee.Print;
END;
```

Then you could define the Contractor subclass.

```
TContractor = OBJECT(TEmployee)
      fContractNumber:  integer;
      fContractDollarAmount:  real;
      PROCEDURE TContractor.Print;  OVERRIDE;
END;
```

You should notice three things about this class definition:

- The TContractor class has been declared to be a descendant of the TEmployee class.
- Only the fields specific to the TContractor class are listed. To be sure, the TContractor class has four fields: fName, fTitle, fContractNumber, and fContractDollarAmount. Since the first two are inherited, however, only the last two need to be listed in this class definition.
- The Print method, which is an inherited method, is listed again. If you wanted the Print method of TContractor to be exactly the same as the Print method of TEmployee, you would not have to list TContractor.Print—it would be automatically inherited. However, since TContractor.Print overrides TEmployee.Print, it does need to be listed, and the keyword OVERRIDE, followed by a semicolon, is written at the end of the method declaration line.

Object Pascal makes one limitation on override methods: An override method must have the same interface as its inherited method. Therefore an override method must have *exactly* the same parameter list as its inherited method, including having the same formal parameter names. Since an override method is supposed to be "equivalent" to the inherited method—only customized for a descendant class—this rule makes sense. However, there are times when it will pose a bit of a problem, especially with initialization methods, which will be dealt with later in this chapter.

A class definition is hardly complete without defining the methods that belong to that class. However, it will be useful to examine Object Pascal instances before exploring the syntax of method definitions.

Instances

Since instances in Object Pascal are similar to records in standard Pascal, let's first look at record instantiation.

Record handles and record instantiations

Standard Pascal allows records to be instantiated a number of different ways: through record variables, through record pointers, and through record handles.

Instantiation through record variables

Records, of course, can simply be instantiated as variables; for example:

```
ThisRecord: RecordType;
```

This line declares an actual record instantiation, named ThisRecord, to be created, which reserves space for ThisRecord in memory.

To reference the fields of ThisRecord, simply use a dot:

```
SomeVariable := ThisRecord.AField;
```

Instantiation through record pointers

Another frequent means of instantiating a record is through the use of a pointer, for example:

```
TYPE
      RecordPtr = ^RecordType;
VAR
      ThisRecord:RecordPtr;
```

When you use pointers, you must first call Pascal's NEW routine to allocate space in memory for the record. Then the fields of the record can be referenced by the special up-arrow notation:

```
NEW(ThisRecord);
SomeVariable := ThisRecord^.AField;
```

Instantiation through record handles

Finally, a third means of instantiating records, and one that is very popular with Macintosh programmers, is through the use of a handle. A handle is a pointer to a pointer and is declared like this:

```
TYPE
      RecordHdl = ^RecordPtr;
VAR
      ThisRecord: RecordHdl;
```

Before the record actually exists, you must call the NewHandle routine. Then you can access the fields of the record through the use of the double indirection symbol:

```
ThisRecord:= RecordHdl(NewHandle(sizeOf(RecordType)));
x:= ThisRecord^^.aField;
```

Using handles and double indirection may look like a lot of extra work, since it does approximately the same job as using pointers. However, handles allow more efficient memory management, and handles are frequently used in place of pointers in Macintosh programming. Since classes are instantiated solely through the use of handles, the rest of the instantiation examples are shown using handles.

It is important to remember that declaring a record handle does not allocate space in memory for a record. Only after the NewHandle procedure is executed is a space in memory created for storing values into the data fields.

Object references

Instances of a class are created in almost the same manner as record handles. If it were exactly the same, you would declare variables that would store the handle like this:

```
EmployeeA: ^^TContractor;      { But it's not really done this way. }
```

Since this handle variable will "refer" to the object instantiation, it is usually called an **object reference variable,** or just an object reference.

Since all object reference variables are handles, Object Pascal lets you take a shortcut. In fact, it forces you to take the shortcut. The shortcut is this: *All double indirection is done for you automatically.* You never see the double up-arrow when dealing with object reference variables. So, the above declaration becomes:

```
EmployeeA: TContractor;          { This is actually how it's done. }
```

EmployeeA is still an object reference variable. In other words, EmployeeA is a handle that points to an instance of class TContractor. In order to create space in memory for an actual object , you must call the Pascal NEW routine:

```
NEW(EmployeeA);
```

Notice that this is not the same NEW routine as in standard Pascal. This Object Pascal NEW routine takes an object reference variable as a parameter, and creates an object for it to refer to through a handle.

Finally, to reference a field of the object , you use a syntax exactly like that for records:

```
SomeStringVariable := EmployeeA.fName;
```

Note that once again, the double indirection is done for you. Otherwise the above line would read:

```
SomeStringVariable := EmployeeA^^.fName;   { This is NOT how it's done. }
```

Even though this is what is actually happening, you never see this in an Object Pascal program—Object Pascal notices that EmployeeA is an object reference variable and automatically does double indirection for you. While this may seem confusing at first, it actually simplifies things greatly. Except for always remembering to call the NEW procedure, you can usually just think of object reference variables as plain variables instead of handles. This also makes code much easier to read.

Method call syntax

The secret of calling methods in Object Pascal can be easily learned by memorizing one sentence: *In Object Pascal, making a method call looks just like referencing a field.* In other words, an object's methods are called the same way an object's fields are referenced—by using a dot.

For example, if you wanted to call the Print method of the EmployeeA object , the call would look like this:

```
EmployeeA.Print;
```

This looks just like referencing a field of a record variable, but actually it is calling the method of an object. Here again you can see why it is so important to begin all field names with a lowercase f. You will find you get used to any residual ambiguity quite quickly, especially when parameters are involved. For instance, if the Print method required parameters, the call might look like this:

```
EmployeeA.Print(true, 7);
```

This line of code would call the Print method of the instance referred to by the EmployeeA reference variable with true and 7 as the input values for the two parameters.

If the method is a function method, as is the YearlyProfit method of the TCompany class, then it is used syntactically just like standard functions:

```
VAR CompanyA:  TCompany;

...
NEW(CompanyA);
SomeIntegerVariable := CompanyA.YearlyProfit;
```

Now that you've seen object reference variables and instances in Object Pascal, you're ready to see how methods are defined.

Method definitions

Once your classes have been defined, you must write the code for each of the methods you have declared. The syntax for such method definitions is also similar to standard Pascal, with a few exceptions.

Remember that a method definition describes the general behavior of a method for an entire class of objects, but when you invoke the method , it operates only on the data of a particular instance. Object Pascal takes care of calling the instance method for you, but when you define the method for the class, you must give the directions stating what you want the instance methods to do. You do this using Object Pascal's `SELF` keyword.

The SELF keyword

Defining a method—that is, writing the code for a method—is syntactically very simple. The opening line of the definition must exactly match the opening line of the method declaration (the one found in the class definition). The opening line is then followed by the body of the method. In the body of the method, you will often need to refer to a field of the class that owns this method. You can refer to the class's fields using the SELF keyword. For example,

```
PROCEDURE TEmployee.Print;

BEGIN
    writeln('My name is ', SELF.fName);
    writeln('My title is ', SELF.fTitle);
END;
```

This method definition is fairly simple. When you send a Print message to a specific object, it will print that object's fName field, and then it will print that object's fTitle field. In other words, the keyword SELF is like a marker that means "the specific instance that this method operates on." The instance that `SELF` refers to is called the **calling instance.**

Let's take an example. Suppose you have created an Employee reference variable:

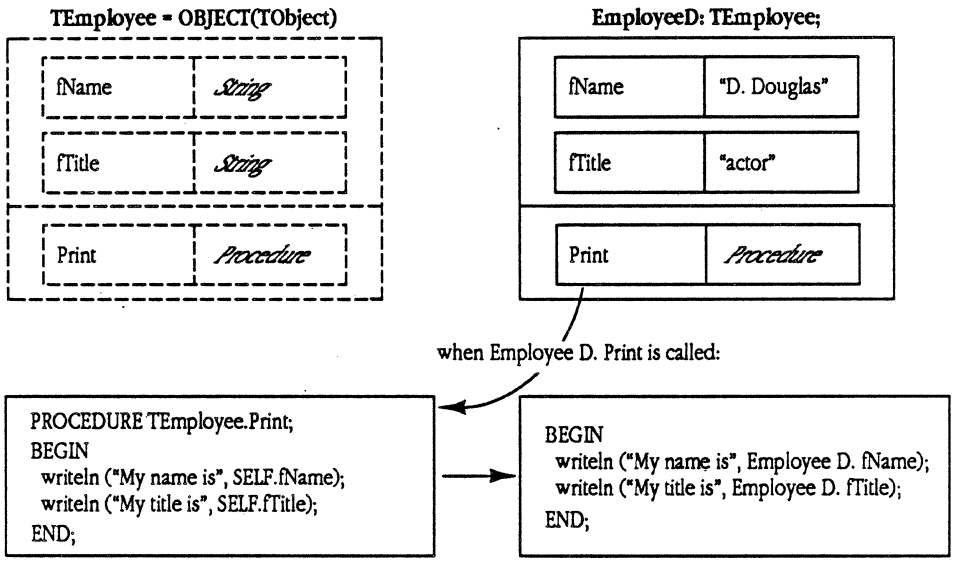
```
VAR  EmployeeD:  TEmployee;
```

and then you have instantiated that variable, and initialized it:

```
NEW(EmployeeD);
EmployeeD.fName := 'D. Douglas';
EmployeeD.fTitle := 'actor';
```

Finally, you invoke its Print method. The Print method executes as if every occurrence of the keyword SELF has been replaced by "EmployeeD."

■ **Figure 3-3** A method as it is defined and as it executes



When a call is made to the Print method of EmployeeD:

```
EmployeeD.Print;
```

EmployeeD is called the calling instance. The keyword `SELF` is always replaced at runtime by a reference to the calling instance.

So, you can see that the keyword `SELF` is the trick that allows you to define a method once, even though it may execute uniquely for each different object you create.

One more thing—the use of the word `SELF` in Object Pascal is optional, though it often makes your code clearer. You can act as if the code part of every method is surrounded by

```
WITH SELF DO BEGIN
  .
  .
  .
END.
```

In other words, every variable or method name is checked by the compiler to see if it belongs to `SELF`. For example, the above Print method could be redefined like this:

```
PROCEDURE TEmployee.Print;

BEGIN
    writeln('My name is ', fName);
    writeln('My title is ', fTitle);
END;
```

The keyword SELF, then, rarely *must* be used. The only situations that require the use of SELF are when you need to pass a reference to the calling instance to another routine or need to assign a reference to the calling instance to a variable. However, always using SELF may help make your code clearer while you are learning Object Pascal.

Methods that call other methods of the calling instance

You've seen how to refer to a field of the calling instance in the method definition using SELF. Making a call to another method of the calling instance works similarly. For instance, here is a possible definition of the Print method of the TSalariedEmployee class:

```
PROCEDURE TSalariedEmployee.Print;
BEGIN
    writeln('My name is ', SELF.fName);
    writeln('My title is ', SELF.fTitle);
    writeln('My salary is ', SELF.fSalary);
    writeln('My weekly paycheck is ', SELF.WeeklyPaycheck);
END.
```

As you can see, the fourth line of this definition looks strikingly similar to the first three lines. The difference is that the first three lines access fields of the calling instance, while the fourth line calls another method (a function method) of the calling instance. In general, function method calls are indistinguishable from field references, which explains the use of a lowercase *f* to begin field names. Procedure method calls also look the same, but of course they return no value and you can distinguish them in context.

Methods that call methods of other instances

As you just saw, calling methods of the same instance is as simple as referring to fields of that instance. Calling methods of other instances is different. (Remember, by convention in Object Pascal, one object should not reference fields of other objects—it should only call their methods.) In order to call methods of another instance, the first instance must have access to the other instance. There are only three ways in which this is possible.

Calling a method of a global object

Imagine you have declared a global object reference variable:

```
VAR globalObject:  TSomeObjectClass;
```

Assuming that this global object reference variable has been instantiated and initialized somewhere in the main program, you can effectively call its methods in any method of any object. For example,

```
PROCEDURE TSampleObject.SampleMethod;
BEGIN
    { Calls to other methods of the calling instance look like this.}
   SELF.OtherMethod;
   { Here's a call to a method of an object instance that is not the}
                                                    { calling instance.}
   globalObject.SomeMethod;
END.
```

Calling a method of a local object

The second possibility for calling another object's methods is having the object be a local variable to the method. Remember, methods are procedures and functions, and as such can have their own local variables—including object reference variables. Using this strategy, you could redefine the TSalariedEmployee.Print method again.

```
PROCEDURE TSampleObject.SampleMethod;
VAR localObject:  TSomeObjectClass;
BEGIN
  { Calls to other methods of the calling instance look like this.}
  SELF.OtherMethod;
  { Before calling any methods of the local object, you must instantiate it.}
  NEW(localObject);
  { Here's a call to a method of the local instance -- not the calling }
  { instance.}
  localObject.SomeMethod;
END.
```

Calling a method of an instance linked by an object reference field

The final possibility for calling other objects' methods is the most common, and the one discussed in Chapter 2—having an object reference field linked to the other object. For example, remember the Company class:

```
TCompany = OBJECT(TObject)
      fCompetitor:  TCompany;
      fName:  Str255;
      fNumberOfEmployees:  integer;
      fYearlySales:  real;
      fYearlyExpenses:  real;
```

```
      FUNCTION TCompany.YearlyProfit:  real;
      FUNCTION TCompany.ReturnName:  Str255;
      PROCEDURE TCompany.Print;
END;
```

The Print method of class TCompany prints not only the fields of the calling instance, but also the Name field of the calling instance's competitor. The Print method does this by calling the name returning method of the instance referred to by the fCompetitor field. The syntax for this method definition is this:

```
PROCEDURE TCompany.Print
BEGIN
    { First, print the values of the calling instance's fields. }
    writeln('Our company's name is ', SELF.fName);
    writeln('The number of happy employees here is:',
SELF.fNumberOfEmployees);
    writeln('Our yearly sales are: ', SELF.fYearlySales);
    writeln('while our yearly expenses are: ', SELF.fYearlyExpenses);
    { Then, print the value returned by another method of }
    { the calling instance.}
    writeln('which means our yearly profit is: ', SELF.YearlyProfit);
    { Finally, print the value returned by a method of the instance }
    { that is referred to by the fCompetitor field. }
    writeln('Our competitor's name is', SELF.fCompetitor.ReturnName);
END.
```

Notice the last line of code in the above example. It makes the following method call:

```
SELF.fMainCompetitor.ReturnName
```

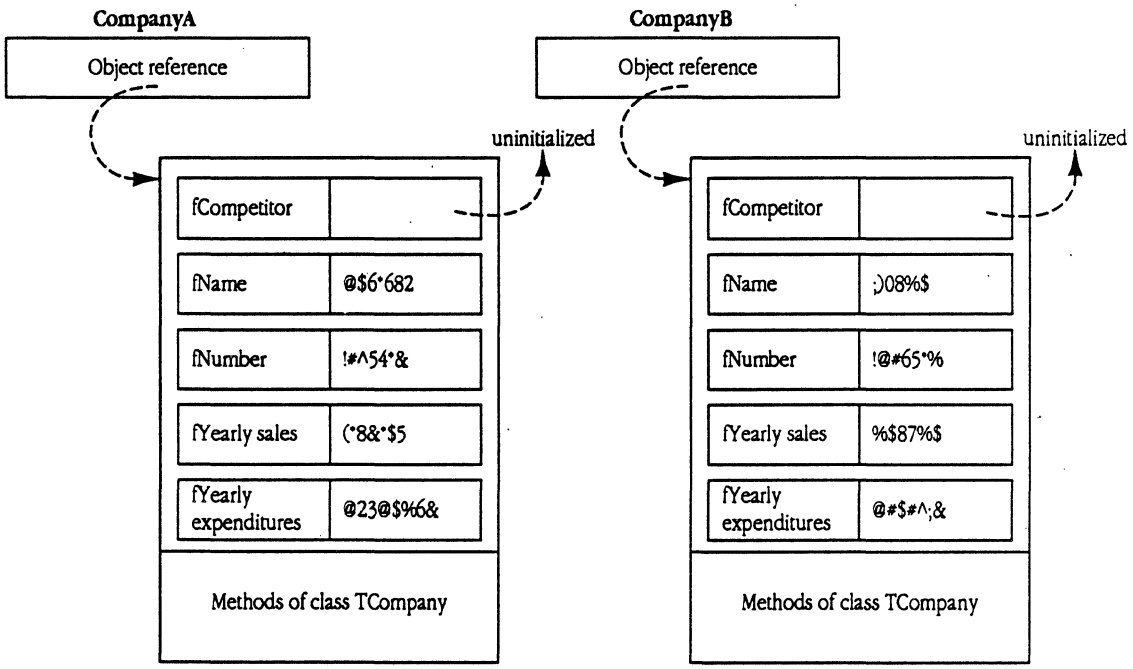
This syntax might seem a bit strange at first. What it means is actually quite simple—it's like referencing a field of a record field of a record. The `SELF` keyword refers to the calling instance. Adding `.fMainCompetitor` makes it refer to the object linked by that field—the calling instance's main competitor object. Adding the "`.ReturnName`" specifies that a call to the ReturnName function method of the main competitor object is to be made.

In order to see this in action, let's create some instances of class TCompany.

```
VAR CompanyA, CompanyB:  TCompany;
BEGIN
    {instantiate the object reference variables }
    NEW(CompanyA);
    NEW(CompanyB);
    { initialize CompanyA's fields }
    CompanyA.fCompetitor := CompanyB; { This is how the one link is }
                                      { established.                }
    CompanyA.fName := 'Andy's Widgets, Inc.'
    CompanyA.fNumberOfEmployees := 100;
    CompanyA.fYearlySales := 5000000.00;
    CompanyA.fYearlyExpenditures := 4000000.00;
    { initialize CompanyB's fields }
    CompanyB.fCompetitor := CompanyA; { This is how the other link is }
                                      { established. }
    CompanyB.fName := 'Mark's Foobat Co.'
    CompanyB.fNumberOfEmployees := 50;
    CompanyB.fYearlySales := 2000000.00;
    CompanyB.fYearlyExpenditures := 1000000.00;
    ...
END.
```

After the first two lines of code have been executed, uninitialized objects have been created in memory.

■ **Figure 3-4** Uninitialized objects in memory



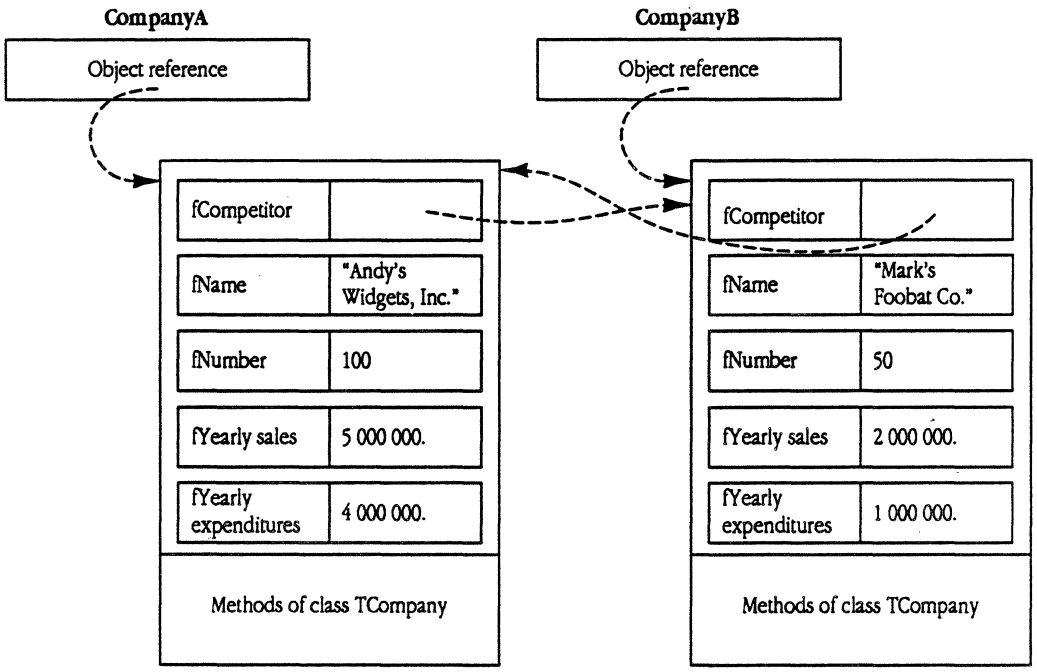
The next ten lines of code initialize these objects. Most of this initialization is straightforward. The only unfamiliar initialization is the establishment of the references between the two objects. If you remember that object reference variables are actually handles, these two lines look like standard Pascal code. Take the first of these as an example:

```
CompanyA.fCompetitor := CompanyB;
```

This line specifies that the fCompetitor field should point to the object referenced by the CompanyB object reference variable.

Once all of the code above has executed, the following situation will exist in memory.

■ **Figure 3-5** Object references and objects in memory



Now, if you call the Print method of CompanyA at the end of the main program above, like this:

```
CompanyA.Print;
```

the line that read

```
writeln('My competitor's name is', SELF.fCompetitor.ReturnName);
```

in the method definition will execute just as this line would during runtime:

```
writeln('My competitor's name is', CompanyA.fCompetitor.ReturnName);
```

When this line is executed, the ReturnName method of the object referenced by the fMainCompetitor field of CompanyA is called, and the result is printed, which is exactly what was wanted.

Override methods

An override method is defined just like any other method, except for the use of the keyword OVERRIDE, just as in the method declaration line.

For example, remember that class TSalariedEmployee is a subclass of class TPaidWeeklyEmployee, and therefore TSalariedEmployee inherits the WeeklyPaycheck method of TPaidWeeklyEmployee. Of course, TPaidWeeklyEmployee was an abstract object class, and its WeeklyPaycheck method was never meant to do anything:

```
PROCEDURE TPaidWeeklyEmployee.WeeklyPaycheck;
BEGIN
    { This method does nothing. It is meant to be overridden. }
END.
```

Defining the override version of this method looks exactly like defining any method, except for the keyword OVERRIDE.

```
PROCEDURE TSalariedEmployee.WeeklyPaycheck;  OVERRIDE;
VAR amountOfCheck: real;
BEGIN
  { Determine the amount of the paycheck, using the calling instance's fields }
  amountOfCheck := SELF.fSalary / 52;
  { Return the result, just as you would for any Pascal function. }
  WeeklyPaycheck := amountOfCheck;
END.
```

Override methods that call the inherited method

Sometimes it is desirable for an override method to make use of the inherited method that it is overriding. Remember from Figure 2-21 that the Print method of the Contractor class overrode the Print method of the Employee class, but still made a call to that inherited method. In Object Pascal, the definition of the override method would look like this:

```
PROCEDURE TContractor.Print; OVERRIDE;
BEGIN
    { First, call the inherited version of Print
    {to print name and title. }
    INHERITED Print;
    { Then, print out the fields specific to this class. }
    writeln('My contract number is ', SELF.fContractNumber);
    writeln('My contract dollar amount is ',
SELF.fContractDollarAmount);
END.
```

You can see that the syntax in Object Pascal for making calls to inherited methods is simple:

```
INHERITED Print;
```

If the method has any parameters, they are listed after the method name, as usual. For example,

```
INHERITED Print(parameter1, parameter2);
```

More about Object Pascal

By now, you've seen all of the syntax of Object Pascal. You know how to define classes and their methods, and declare and instantiate object reference variables. Syntactically, you're ready to start creating Object Pascal programs. However, there are a few use and convention issues that you should be familiar with before diving into your first MacApp program.

Creating and freeing instances

Earlier in this chapter, the class TObject was mentioned. The TObject class has no fields and only a few utility methods. These utility methods are helpful enough that it is desirable for all classes to have them. This is accomplished easily: Just make your highest ancestor objects descend from class TObject. For example, in the Employee class hierarchy, TEmployee is the superclass—it has no superclass itself. In this case, all you need to do is make TEmployee a subclass of TObject. Since all other classes in that example descend from the TEmployee class, all classes therefore descend from TObject as well. As a result, all classes inherit the methods of class TObject.

You already know how to allocate memory for an instance—use the NEW routine on an object reference variable. To deallocate the memory of an instance, you can use the **Free** method. The Free method is a method of TObject and therefore can be called by any instance, if you've declared your hierarchy properly. You can override the Free method so that objects of a given type can perform cleanup tasks of their own when they are about to be deallocated.

Initialization methods

Immediately after instantiating an object reference variable, it is a good idea to initialize the object . Typically this is done with an initialization method—a method that takes as many parameters as the class has fields, and initializes each field to one of the parameters.

For example, the TEmployee class might have the following initialization method.

```
PROCEDURE TEmployee.Initialize(name, title: Str255);
BEGIN
    { Initialize each field to the appropriate parameter. }
    SELF.fName := name;
    SELF.fTitle := title;
```

```
END;
```

If you have declared an object reference variable, for example EmployeeD, to be of class TEmployee, then after you instantiate it you can call its initialization method:

```
EmployeeD.Initialize('D. Douglas', 'actor');
```

Like the Print method example, you might think it would be nice to give each Employee class an override version of this method. The override version could call the inherited version to initialize the first two fields, and then initialize the rest of the fields itself.

For instance,

```
{ NOTE: For example only -- this is NOT how it's actually done! }
PROCEDURE TContractor.Initialize(name, title: Str255;
                                 number: integer;
                                 amount: real);  OVERRIDE;
BEGIN
    { Call the inherited method -- the method of TEmployee }
    INHERITED Initialize(name, title);
    { Initialize the remaining two parameters. }
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;
END;
```

Unfortunately, Object Pascal requires that the method declaration line not be changed for override procedures. As you can see in the above example, there were two parameters added to the method declaration line. This is not legal and the above override procedure would not compile.

Since initialization methods almost always require varying numbers of parameters, the result of this restriction is that initialization methods are not override methods. Each class has its own initialization method. For example, the two classes above might have these two initialization methods:

```
{ NOTE: This example shows how initialization is actually done. }
```

```
PROCEDURE TEmployee.IEmployee(name, title: Str255);
BEGIN
    SELF.fName := name;
    SELF.fTitle := title;
END;

PROCEDURE TContractor.IContractor(name, title: Str255;
                                  number: integer;
                                  amount: real);
BEGIN
    SELF.fName := name;
    SELF.fTitle := title;
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;
END;
```

As you can see, the convention for naming initialization methods is to begin them with an uppercase *I*, followed by the name of the class.

Even though you cannot have IContractor call an inherited initialization method (because there isn't one), you can do something almost as good. Since class TContractor inherits all of the methods of class TEmployee, it must also inherit IEmployee. Therefore, you can have IContractor call IEmployee directly—almost as if you were calling an inherited initialization method:

```
PROCEDURE TContractor.IContractor(name, title: Str255;
                                  number: integer;
                                  amount: real);
BEGIN
    SELF.IEmployee(name, title);
    SELF.fContractNumber := number;
    SELF.fContractDollarAmount := amount;
END;
```

In this manner, making separate initialization methods that are not override methods for each class prevents you from having to duplicate any code.

Privacy between instances

As you've seen a few times, the method of one object can call the method of another object. Under the strict application of convention, this is the only way one object should communicate with another. However, there are times when you will want a method of one object to have access to information from another object and there will be no method of the second object that returns that data. In cases like this, it would be best to add a method to the second object's class that did return that data, but this is not always possible.

It is legal in Object Pascal syntax for the method of one object to reference a field of another object. For example, the method definition for TCompany.Print could include something like this:

```
PROCEDURE TCompany.Print;
BEGIN
    { Print the value of the fName field of the competitor instance. }
    writeln('My competitor's name is', SELF.fCompetitor.fName);
END.
```

Here, you can see that the method is accessing the fName field directly. This ability exists in Object Pascal, but use it sparingly! Any data field may have a number of other fields and methods dependent on it at a given moment. If you modify a field without notifying all of its dependent fields and methods, you significantly increase your chances of destroying data structures dependent on that field and are more likely to crash the program.

Object reference variables

Object reference variables must be declared to be of a specific class. For example,

```
VAR SomeEmployee:  TEmployee;
```

SomeEmployee is an object reference variable, and is a member of class TEmployee. Therefore, it must reference an instance of class TEmployee. However, in Object Pascal, SomeEmployee is also allowed to reference an instance of any class that is descended from TEmployee. For instance, imagine you have also declared a variable EmployeeA:

```
VAR EmployeeA:  TContractor;
```

In the main body of the program, after instantiating and initializing EmployeeA, you can make the following assignment:

```
SomeEmployee := EmployeeA;
```

Now, SomeEmployee is referencing an instance of class TContractor. Since TContractor descends from TEmployee, this is legal. SomeEmployee now has access to all of the fields and methods of TContractor, as well as the fields and methods of TEmployee. For example, it is now legal to reference SomeEmployee.fContractNumber.

The real power of object-oriented programming comes from what happens when you access a method of SomeEmployee: The correct method will always be called, regardless of whether SomeEmployee references an instance of the class that originally defined it or an instance of one of its descendant classes.

For example, if you now call SomeEmployee.Print, the override Print method belonging to the TContractor class is the version that is called—not the original Print method belonging to the TEmployee class. This means that as late as runtime your object-oriented program is checking which instance is the calling instance and is making sure the right version of the method is called.

Summary of Object Pascal syntax

- Classes are defined in the TYPE declaration part of a Pascal program. These definitions include the class name (beginning with an uppercase *T* by convention), the type name OBJECT, the superclass name in parentheses, and the field declarations and the method declarations (including overridden methods followed by the keyword `OVERRIDE`.

```
TClassName = OBJECT(TObjectClassAncestor)
      {field declarations}
      {method declarations}
END;
```

- Object fields (beginning with a lowercase f by convention) are declared just like record fields.

```
fFieldName:  fieldType;
```

- Methods are declared as normal Pascal procedures and functions. Their names include the name of their class. If it is an overridden method, the keyword OVERRIDE ends the declaration line.

```
PROCEDURE TClassName .TMethodName;
FUNCTION TClassName .TMethodName: ResultType;

PROCEDURE TClassName .TMethodName(parameter list);
FUNCTION TClassName .TMethodName(parameter list): ResultType;

PROCEDURE TClassName .TMethodName; OVERRIDE;
FUNCTION TClassName .TMethodName: ResultType; OVERRIDE;

PROCEDURE TClassName .TMethodName(parameter list); OVERRIDE;
FUNCTION TClassName .TMethodName(parameter list): ResultType; OVERRIDE;
```

- Methods are generally considered **public**—that is, they can be accessed by other objects' methods—while fields are generally considered **private**—that is, other objects should not access them directly. Though just a convention and not enforced by Object Pascal, strict adherence to this rule leads to more robust code and is strongly recommended.
- Object Pascal does all object-related double-indirection. Even when an object reference variable is declared, no indirection is expressly used.

  ```
  ObjectReferenceVariableName : ObjectClass;
  ```

  The function NEW is still necessary, however, since object reference variables are still handles.

  ```
  NEW(ObjectReferenceVariableName);
  ```

- Fields and methods are both accessed with a dot. No indirection is ever needed.

  ```
  ObjectReferenceVariableName.fFieldName
  ObjectReferenceVariableName.MethodName
  ```

- Methods are defined (actually coded) much like standard Pascal procedures and functions. The title line must match the method declaration line *exactly* (including parameters and OVERRIDE). The keyword SELF is used to refer to the object that called this method. SELF is always optional.
- Inherited methods can be called in an overridden method by prefixing the line of the call with the keyword INHERITED.

  ```
  INHERITED InheritedMethodCallName;
  ```

Chapter 4 Introduction to MacApp: Organization

BY NOW, YOU SHOULD HAVE A GOOD IDEA of what object-oriented programming is all about, and should be ready for an in-depth look at MacApp. MacApp is nothing more than an extensive library of code. As with any large system, having an understanding of the whole helps to make sense of the parts. This chapter and the next present an overview of MacApp. This one renders a still-life portrait of MacApp; the next chapter shows the engine in motion—how the user's actions are interpreted and handled while the application is running.

The still-life in this chapter offers three different perspectives on how MacApp is organized:

- the physical location of files on disks
- the hierarchy of inheritance among the classes
- the way MacApp's objects represent the concepts of the Macintosh user interface

Knowing the physical layout of the files helps you include all the necessary code; it also helps you track down the location of any piece of code you want to investigate more thoroughly. Knowing the ancestry of a particular object tells you what fields and methods it has in common with other objects. And knowing how the different objects in MacApp provide metaphors for the concepts of the Macintosh world allows you to understand why MacApp is put together the way it is, giving you the power to manipulate the Macintosh to create effective applications. ■

Source code: Organization of the files

MacApp is written mostly in Object Pascal, organized into units. Technically, a **unit** is a piece of code compiled separately from all the rest; conceptually, all the code compiled into one unit should have something in common. For instance, all the code related to printing might be compiled into a single unit. Once you compile all the units separately, you then have to link them together—a single unit by itself is not executable.

MacApp is divided into 19 units. Many of the units must be included in every application; others are optional, depending, for instance, on whether you want to use the debugger or take advantage of MacApp's printing or text-editing features. All of MacApp's units have names that begin with an uppercase *U*. Table 4-1 lists these units. The table gives a thumbnail description of each unit, mentions the circumstances under which it should be included, and indicates how much of your attention it requires. Many of these units will be of no concern to you, either because their use is optional or because their routines and data are used only by the code in the MacApp library.

■ **Table 4-1** MacApp units

Unit	Responsibilities	When to include	When it needs your attention
UAssociation	Associating one object with another	Always	Rarely
UBusyCursor	Changing cursor shape when the application is busy	Always	Rarely
UDialog	Defining view classes typically used in dialog-style windows	Optional	Optional
UFailure	Handling failures	Always	Sometimes
UGridView	Displaying arrays of information	Optional	Optional
UInspector	Implementing the inspector	Only when debugging	Never
UList	Maintaining lists of objects	Always	Optional
ULoMem	Defining access to low memory globals	Always	Never
UMacApp	The core of MacApp's functions	Always	Always

(continued)

■ **Table 4-1** MacApp units (continued)

Unit	Responsibilities	When to include	When it needs your attention
UMacAppUtilities	Global routines and data	Always	Always
UMemory	Memory management	Always	Always
UMenuSetup	Menu maintenance	Always	Always
UObject	Defining the root superclass of all other objects	Always	When creating custom objects
UPatch	Implementing patches to the Macintosh ROM	Always	Never
UPrinting	Printing	When your application requires printing	When customizing
UTEView	Text editing	Optional	Optional
UDebug	MacApp debugger	Debugging	Never
UViewCoords	32-bit coordinate support	Always	Rarely
UTranscriptView	Debugger window	Debugging	Never

Each unit consists of at least two files: one interface file and one or more implementation files. The interface files, which are always much shorter than the implementation files, contain only declarations and type definitions. The implementation files contain the code that implements these declarations. This means that if you want to take a quick look at what parameters a particular method call requires, you only need to look at the short interface file, in which the method is declared. Each interface file also lists all the implementation files associated with it, since they all need to be compiled together. The filenames in MacApp adhere to the conventions described in Table 4-2.

■ **Table 4-2** Naming conventions for MacApp files

File type	Naming convention	Examples
Assembly	U*filename*.a	UMacApp.a
Interface	U*filename*.p	UMacApp.p
Implementation	U*interface.implementation*.p	UMacApp.incl.p *or* UMacApp.TApplication.p

In most cases, a unit's implementation portion is short enough to be put into a single file. In that case, the implementation section's name is traditionally just "incl," which stands for "include file #1." However, when the implementation is too long to be managed easily in a single file, it is broken up across a number of files, with each given a mnemonic name. The most notable example of a unit broken into a large number of files is UMacApp, which is broken up into about a dozen files, each named after the most important thing defined in the unit, usually an object class. So, the file UMacApp.TApplication.p is the implementation portion of the UMacApp unit that defines the class named TApplication.

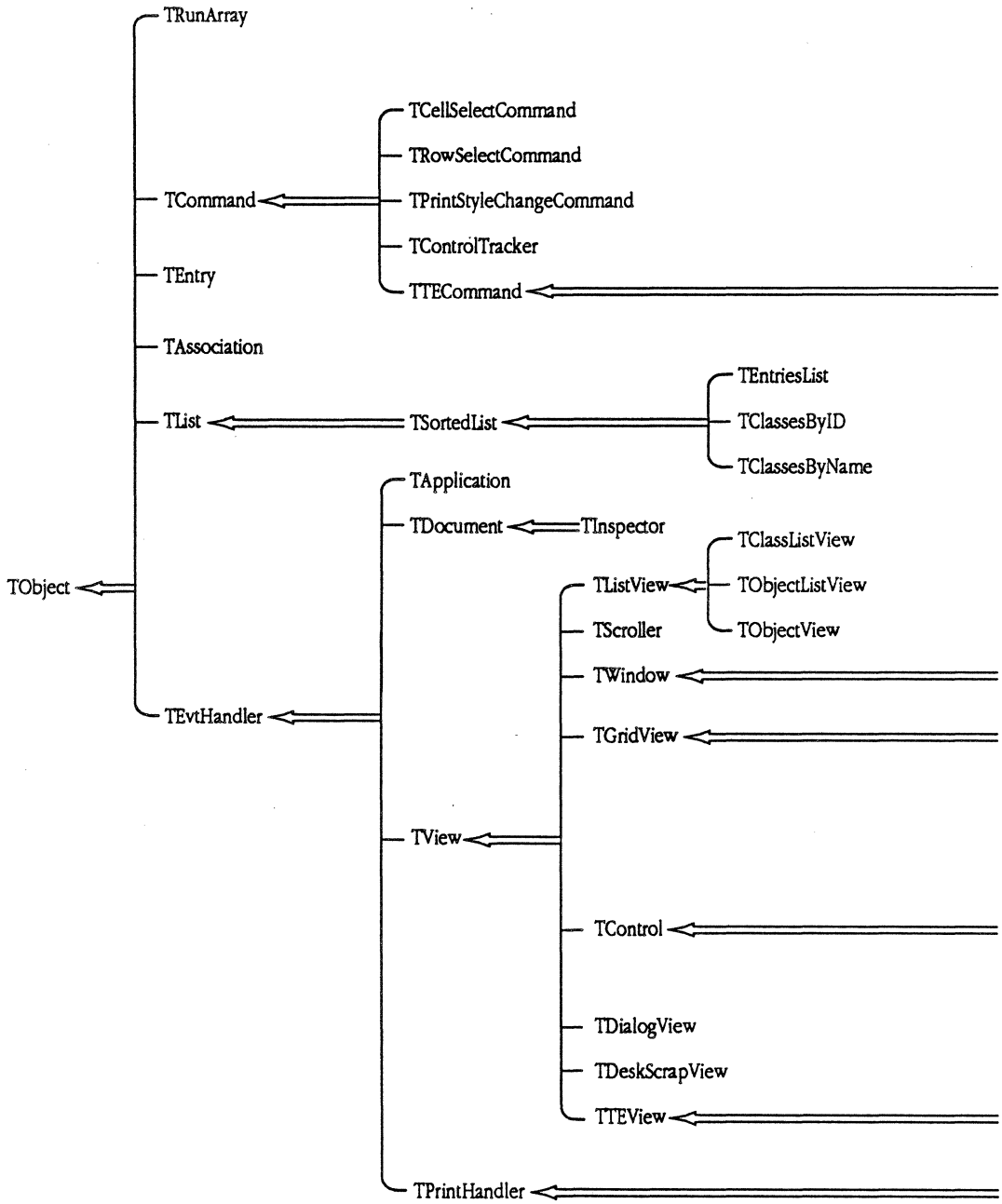
Each unit contains code related to a particular aspect of MacApp. A unit may contain any proportion of Object Pascal and standard Pascal. Though MacApp is an object-oriented application framework, much of the MacApp library consists of routines written in standard Pascal to make your application run more efficiently.

To summarize, the 60 or so files in the MacApp library belong to 19 units, organized according to function. The name of each file reflects its unit name and whether the file contains the interface or the implementation.

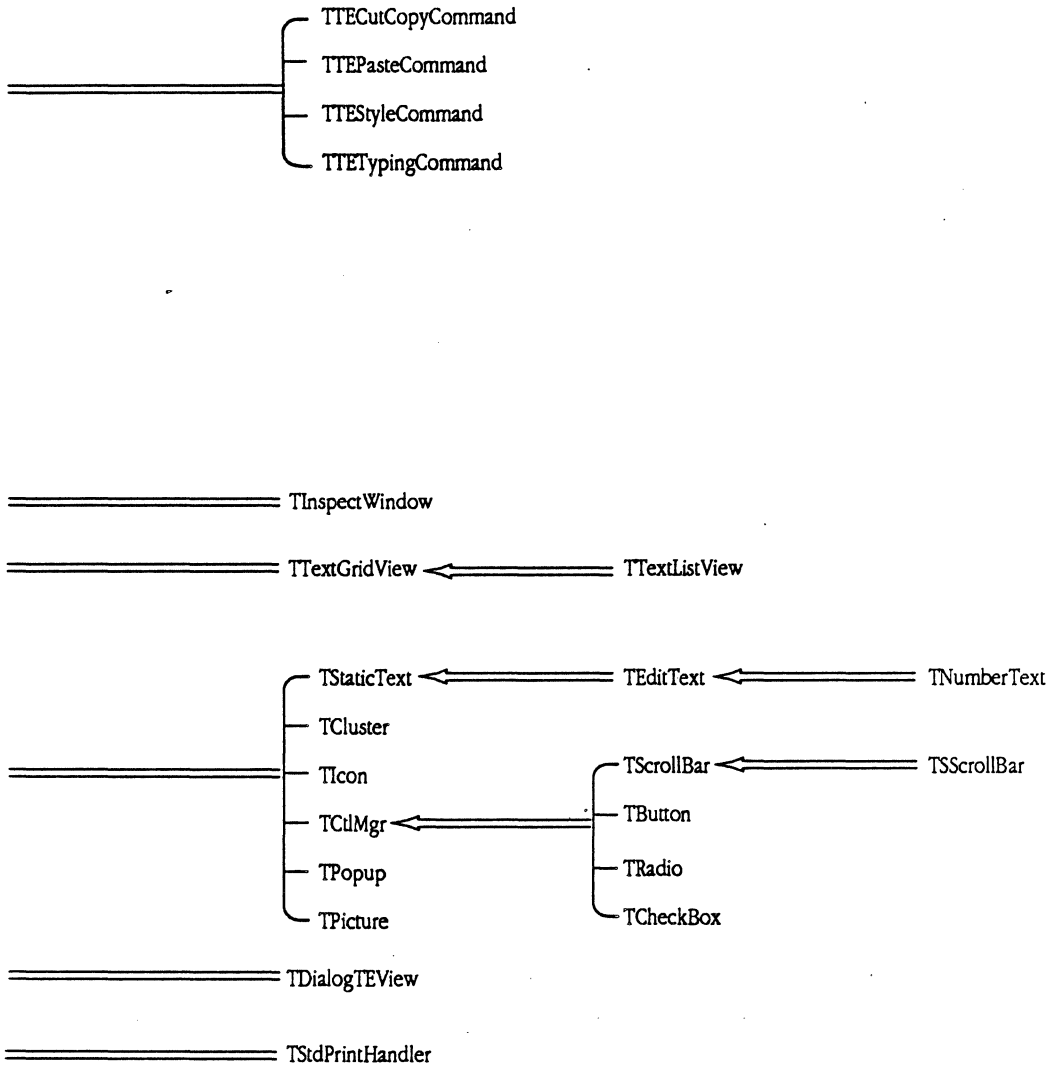
Ancestry: Organization of the classes

MacApp defines 54 different classes of objects; fortunately, you only have to understand about a dozen of these to get a good overview of the library. The other three dozen or so classes are less important to understand immediately, either because you will never use them directly (though they are used by other pieces of the MacApp library) or because their purpose is highly specialized. For instance, almost half of the classes are devoted to specific kinds of views, and though you will probably use some of these classes quite frequently, you don't need to know about them all to get an overview of MacApp.

■ **Figure 4-1** The ancestry tree for the classes in MacApp



■ **Figure 4-1** The ancestry tree for the classes in MacApp (continued)



There are two ways of analyzing these classes: by inheritance and by functional design. Inheritance tells you which methods and fields different classes may share, but it does not tell you how the objects were intended to work together. This section discusses inheritance only; the next section discusses functional design (which is also called **architecture**).

Figure 4-1 depicts the ancestry tree for the different classes. The object class **TObject**, from which all the others descend, is all the way to the left. TObject has six immediate subclasses: **TRunArray, TCommand, TEntry, TAssociation, TList,** and **TEvtHandler.** The uppercase *T* at the beginning of each of these names indicates that it is an object *type* (another way of saying *class*), not an actual instance of a particular class. Remember, when you see a name beginning with *T,* it is a cookie cutter, not the cookie itself. In any case, the arrows in Figure 4-1 point from the subclasses toward the superclasses.

As you look at this chart and read about classes and subclasses, remember that subclasses are specialized versions of their superclass. So when you see that TWindow is a subclass of TView, translate that into the words "TWindow is a specialized version of TView."

Each of these objects is described more completely in the next section on architecture. Here, each object is presented only in its sketchiest outline, so you can see how it fits into the hierarchy. Most of the objects described in this section are objects that you will use repeatedly. You can "use" objects in two ways: by creating instances of them or by creating new subclasses of them. Classes that are never used to create objects are called **abstract classes**. They are used only to create subclasses that will share all the methods and fields inherited from their parent.

Higher-level classes

There are four commonly used classes at the top of the ancestry tree: TObject, TList, TCommand, and TEvtHandler.

- **TObject.** TObject is an abstract class, the ultimate superclass of every other class. The main purpose of TObject is to bestow upon all its subclasses a very few abilities all objects should have in common, namely making clones of itself and deallocating any memory it might have required so you can delete it. You will probably never create instances of TObject itself. Rather, both you and MacApp will use it to create more useful subclasses and then will create instances of *those* classes.

TObject has three important subclasses: TList, TCommand, and TEvtHandler.

- **TList.** (Ancestry: TObject ← TList) An instance of the TList class implements a simple list of objects and is used throughout the MacApp library to store such lists. Using a list of object instances is a common way of performing repetitive actions, since you can have each instance in the list perform some operation on itself. For example, in a chess program, you could keep a list of all the instances that represent chess pieces; if you wanted to implement a command that would display the location of every chess piece, you could simply go down the list and execute a Report method for every instance in the list.
- **TCommand** (Ancestry: TObject ← TCommand) The TCommand class is one of the most important to understand, and also one of the most difficult to explain. While it is easy to picture a window or a scroll bar, it is hard to visualize a command. But if you remember that an object is just some data and the methods that manipulate that data, the concept is not difficult at all. An instance of the TCommand class has fields that keep track of the effects of a command and methods to perform the command, to track the mouse, and perhaps to undo and redo the command.
- **TEvtHandler.** (Ancestry: TObject ← TEvtHandler) Of TObject's three subclasses, TEvtHandler is by far the most important, both in terms of how often you will use it and in terms of understanding how MacApp works. Well over two-thirds of the classes defined in MacApp are subclasses of TEvtHandler, and they are conceptually important classes—the ones that represent windows, documents, and the application itself.

  TEvtHandler is an abstract class, the superclass of the many object instances that have to deal with events. The word *event* has a special meaning in the Macintosh world, including almost anything that might occasion a response from an application, ranging from user actions like pressing a key on the keyboard to system-generated requests to refresh a region on the screen.

Much of the code that you write yourself will be written to implement event-handling classes. The most important subclasses of TEvtHandler are TApplication, TDocument, and TView.

- **TApplication.** (Ancestry: TObject ← TEvtHandler ← TApplication) The TApplication class implements the main event loop and directs events to the objects responsible for handling them. The application class itself handles some events, if they affect the application as a whole. To get an idea of the sorts of things the application class is responsible for, imagine an application with no windows open. Any commands that can still be executed (including Open, New, About, or Quit) are handled by an instance of the application object. There is one, and only one, instance of a TApplication class in each application.

- **TDocument.** (Ancestry: TObject ← TEvtHandler ← TDocument) The TDocument class represents a collection of data (which is usually stored in a file). For example, a word processor uses documents consisting of formatting commands and text; a drawing program might use bitmapped documents; and other programs might use several different types of documents at once. Documents are generally responsible for commands that directly affect the data, such as the Save command.
- **TView.** (Ancestry: TObject ← TEvtHandler ← TView) The TView class is responsible for everything your application displays on the screen, except the menus. Views are used to render a document's data. For example, a spreadsheet application may have two views of the same data—tabular and graphic. TCommand, a subclass of TView, is generally responsible for actually handling commands that a user might enter by clicking and dragging in the content region of a window; however, TView implements the highlighting and selection of screen items *after* TCommand sends it that message. The great majority of TEvtHandlers are subclasses of TView.

Subclasses of TView

Since the Macintosh provides a highly visual environment, the TView class includes a diverse group of classes among its subclasses.

- **TListView.** (Ancestry: TObject ← TEvtHandler ← TView ← TListView) TListView is a subclass of TView customized to simulate the functions of the List Manager, implementing simple scrolling lists similar to the one found in the Standard File dialog box.
- **TScroller.** (Ancestry: TObject ← TEvtHandler ← TView ← TScroller) The TScroller class calculates coordinate translations to create the illusion of scrolling through a document.
- **TWindow.** (Ancestry: TObject ← TEvtHandler ← TView ← TWindow) The TWindow class represents Macintosh windows and contains a window's subviews. Since TWindow is a subclass of TEvtHandler, it inherits many fields and methods for intercepting and handling events; and since it is a subclass of TView, it inherits fields and methods for drawing on the Macintosh screen. TWindow overrides some of these, and has extra attributes that allow it to store and display a title and the other parts of a window. It implements standard window manipulation such as resizing, zooming, moving, and tracking mouse clicks in the close box.
- **TGridView.** (Ancestry: TObject ← TEvtHandler ← TView ← TGridView) The TGridView class implements the display of items in a grid, as you might see them in a spreadsheet application. TGridView will display data in variable-sized columns and rows, and it allows insertion or deletion of entire columns or rows.

- **TControl.** (Ancestry: TObject ← TEvtHandler ← TView ← **TControl**) TControl is also a subclass of TView. TControl is an abstract class (one for which you must create subclasses in order to obtain useful instances) that represents the MacApp notion of a control. MacApp controls include Control Manager controls implemented by the subclass TCtlMgr such as radio buttons, push buttons, check boxes, and scroll bars. TControl implements non-Control Manager controls as well, either those of your own creation or as defined in the UDialog unit. The TControl class also contains utility procedures that enable or disable controls, highlight them and do mouse tracking.
- **TDialogView.** (Ancestry: TObject ← TEvtHandler ← TView ← TDialogView) Interestingly, MacApp does not use the Macintosh Dialog Manager at all. Instead, the TDialogView class duplicates the Dialog Manager's functions, creating the illusion (with the help of some other classes) that a MacApp window is a Dialog Manager dialog box.
- **TDeskScrapView.** (Ancestry: TObject ← TEvtHandler ← TView ← TDeskScrapView) As you might guess from its name, TDeskScrapView is a subclass of TView customized to show clipboard data in a MacApp window. The methods in TDeskScrapView will display PICTs or string data; to display customized data from your application you may need to create a specialized subclass of TDeskScrapView.
- **TTEView.** (Ancestry: TObject ← TEvtHandler ← TView ← TTEView) The letters *TE* stand for TextEdit, the set of Macintosh ROM utilities that make a rudimentary text editor relatively easy to provide. TTEView is a class that displays and manipulates text.

Summary of the ancestry tree

TObject is the superclass of all objects. It has three important subclasses: TList, which manages lists of objects; TCommand, which provides a framework for doing and undoing commands; and TEvtHandler, whose subclasses populate most of the rest of MacApp. TEvtHandler, in turn, has three important subclasses: TApplication, TDocument, and TView. TView and its subclasses make up a little over half of the classes in MacApp.

Architecture: Organization of the concepts

Now that you have the pieces of MacApp in front of you, it's time to put the puzzle together. Every application has three major sorts of operations: storing data, displaying information, and executing commands. How does MacApp accomplish these operations?

Storing data

Almost every application needs a metaphorical scratch pad where it can jot down information and perform calculations. In the MacApp world, this scratch pad is called the **document**. The English word *document* generally refers to a piece of paper with text on it. But in computing, a document can store any sort of data. For instance, in a graphics program it could store shapes; in a word processor, it could store text and formatting information; in a spreadsheet application, it could store numeric information; or in a music program, it could store data concerning pitch, timbre, and rhythm.

A program could also have a document that stored more than one kind of data—for instance, text and graphics—or a single program could have two (or more) different types of documents. Many Macintosh programs allow more than one document to be open at a time, and often allow documents of different types to be open at the same time.

How does MacApp implement documents? MacApp provides a class called TDocument, which is an immediate subclass of TEvtHandler. In addition to the inherited **event-handler fields**, it has fields that record the title, creator, type, and date of the document. MacApp's design requires each document instance to take initiative for getting itself displayed. Thus, it has methods that create the views through which it will be displayed and fields that keep track of all the views associated with the document. Since MacApp cannot anticipate how you want to display the data or what sorts of windows and views you might want, these methods are empty in the MacApp library. You must override them to provide your own custom definitions.

Exactly how each sort of document stores information is up to you. Storage can be in memory or on disk. Applications often start by reading a document off the disk, keeping as much of it in memory as RAM allows, and then writing it back to the disk at the user's request. When you design your application, you must choose how you want to represent your data and where you want to store it. Since most applications require documents to be read from and written to the disk, the TDocument class also has fields for storing information about the file associated with a particular document, methods for determining the necessary disk space, and methods for reading and writing data. Again, since MacApp cannot determine what sort of information you will be writing to the disk, many of these methods require you to override them and provide the actual code yourself.

In addition to reading and writing disk files and creating and maintaining the views and windows associated with them, a document's methods are responsible for handling any requests from the user that affect the document. Every event-handler class has a method called **DoMenuCommand.** The document's DoMenuCommand method will call the appropriate methods in response to the user's choice of commands such as Save, Save As, and Revert to Saved. By overriding this method, you can allow your documents to handle other commands you might care to implement. Each event handler also has a DoSetupMenus method, which enables the menu commands for which that handler is responsible.

You never create instances of the TDocument class directly; instead, you create one subclass of TDocument for each different sort of document your program uses. You then customize each subclass with fields for the data structures your application needs, and implement methods to maintain those fields. You might also add fields that reference related documents and views, and you might override and customize all the empty methods responsible for reading, writing, and so on. Then, whenever the user opens an old document or creates a new one, the MacApp code you have customized will dynamically create new instances of these classes.

Displaying information

The Macintosh has a rich visual interface, which means the task of creating images on the screen is particularly important. Programmers who have written software for the Macintosh *without* MacApp know a system of displaying images via grafPorts, bit maps, and bit images. Fortunately, MacApp overlays a simpler architecture on top of this old, complex one. Eventually you will have to know quite a bit about QuickDraw, the set of ROM routines that actually put images on the screen. But this introduction assumes no such knowledge. (If you do know about QuickDraw, please put that knowledge aside for the next few pages.)

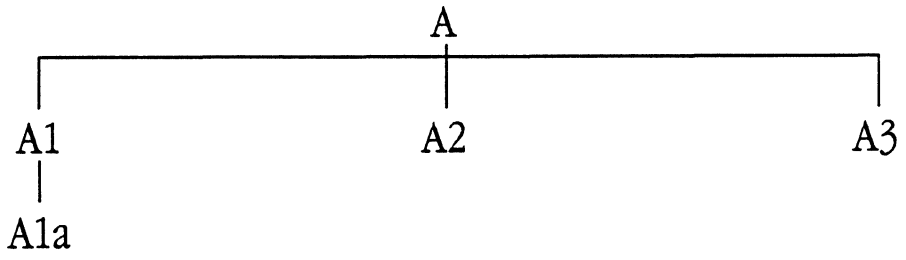
In MacApp, everything your application displays on the screen (except the menu bar) is handled by an instance of some subclass of the class TView, which represents a 32-bit visual coordinate system. TView (Ancestry: TObject ← TEvtHandler ← TView) is an abstract class, so you seldom, if ever, create instances of it directly. Because TView is an event handler, it inherits fields and methods that make handling user events relatively easy.

Hierarchy of views

Each view has **object reference fields** that record information about its relationship to other views. All of these relationships together form a so-called **hierarchy of views.** In MacApp's terminology, each view instance may have one superview, and any number of subviews. Subviews may themselves have other subviews, resulting in a whole tree of views, as depicted abstractly in Figure 4-2. (Less abstract depictions follow later in this chapter.)

△ **Important** It's important to note that related views do not have the same type of relationship with each other that classes have with their superclasses and subclasses. Rather, the phrase *hierarchy of views* refers only to the fact that a view may contain subviews or be part of another superview. These relationships may be thought of like a "family tree" or hierarchy; however, since views are instances, not classes, they *do not* inherit fields or methods from each other.. △

■ **Figure 4-2** Tree of views

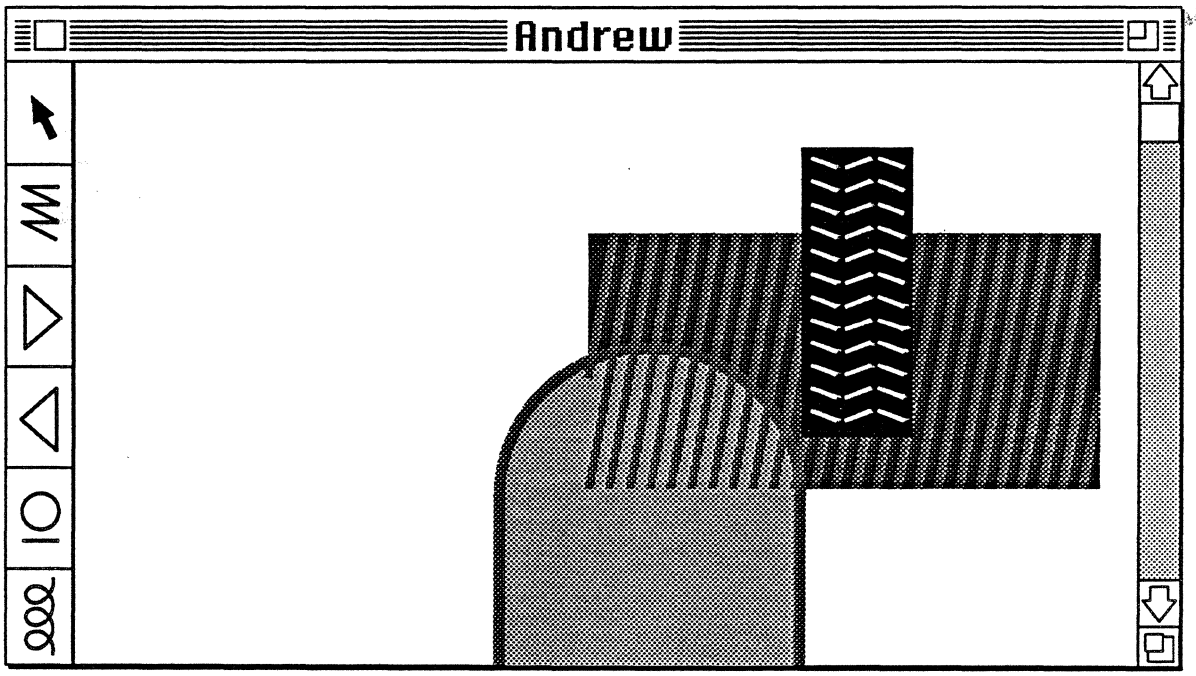


For now, don't try to picture how this tree of views would look on a Macintosh screen. It is enough if you realize that there can be a hierarchy of views. For instance, in the figure above, view A has no superviews, but it has three subviews, A1, A2, and A3. Note that the terms *subview* and *superview* are relative: Just as you can be the child of one person and the parent of another, so one view can be a subview in one case and a superview in another. In Figure 4-2, for example, A1 is the subview of A, but it is the superview of A1a. Each view has its own coordinate system that you define when it is created.

The TView class has fields that keep track of the superview and the subviews, the location and the size of the view, and a pointer to the object responsible for printing it. It has methods that translate coordinates from one view to another, that maintain the lists of subviews and references to superviews, and that can perform actions on all the views or on one view that meets particular criteria. It also has methods for opening, closing, and activating views. (**Activating** takes place when the view's window is brought to the front.) It has other methods for managing the location and size of the view, for drawing and updating the contents of the view (including highlighting the current selection), for handling mouse clicks, and for exporting and importing data to and from the Clipboard. Though the TView class is itself never used to create actual objects, these fields and methods are passed down to all its subclasses.

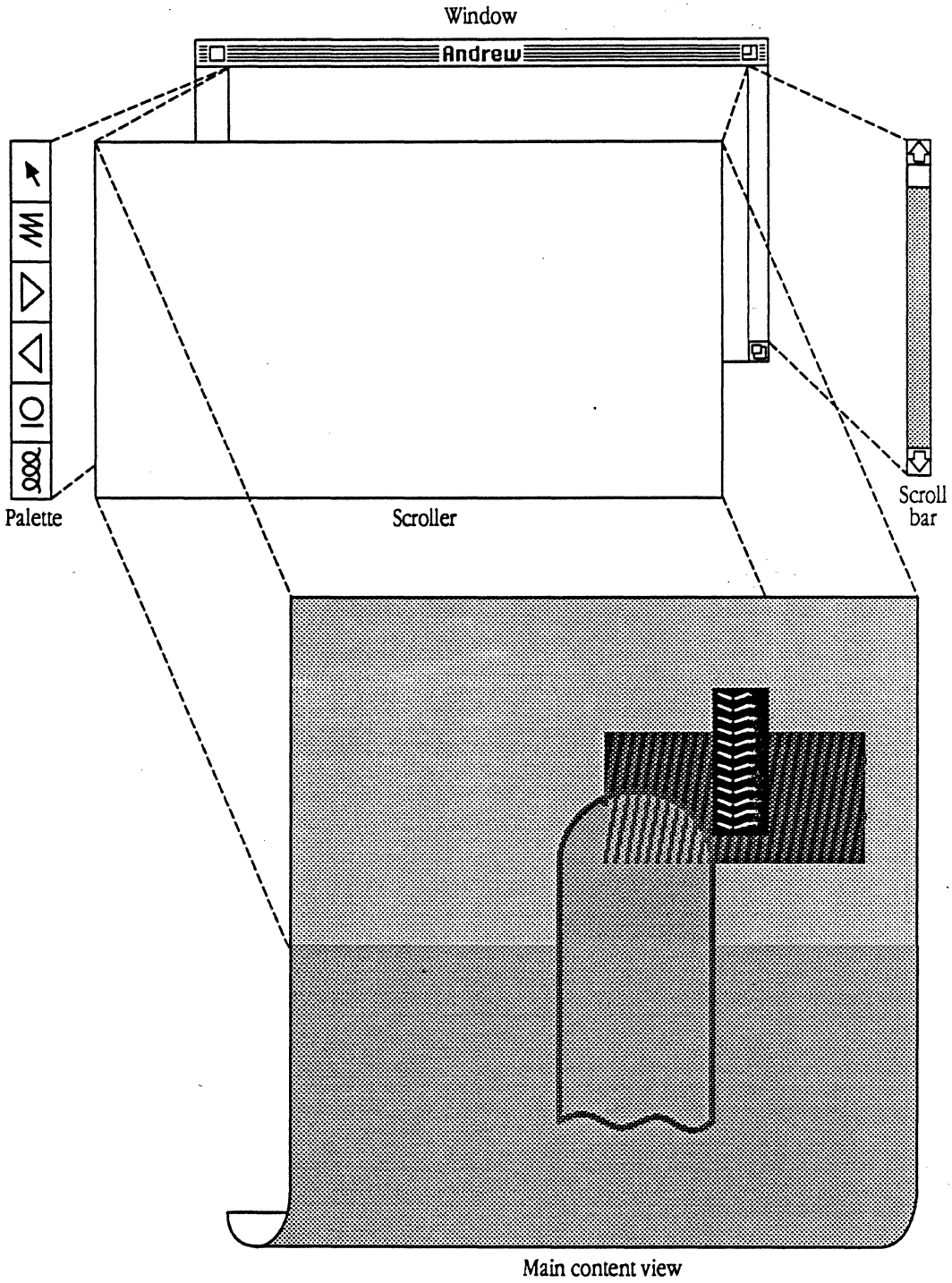
In the MacApp world, everything on the screen except the menu bar appears in a window. Thus, the natural place to begin a discussion about the architecture of views is with the window. A window is at the top of the tree of view instances; in other words, window instances have no superview. The contents of windows can be very simple, displaying only a static image in a fixed-size view with no scroll bars, or they can be very, very complex. Figure 4-3 illustrates a window with fairly complex contents.

■ **Figure 4-3** A fairly complex window with one scroll bar and a palette



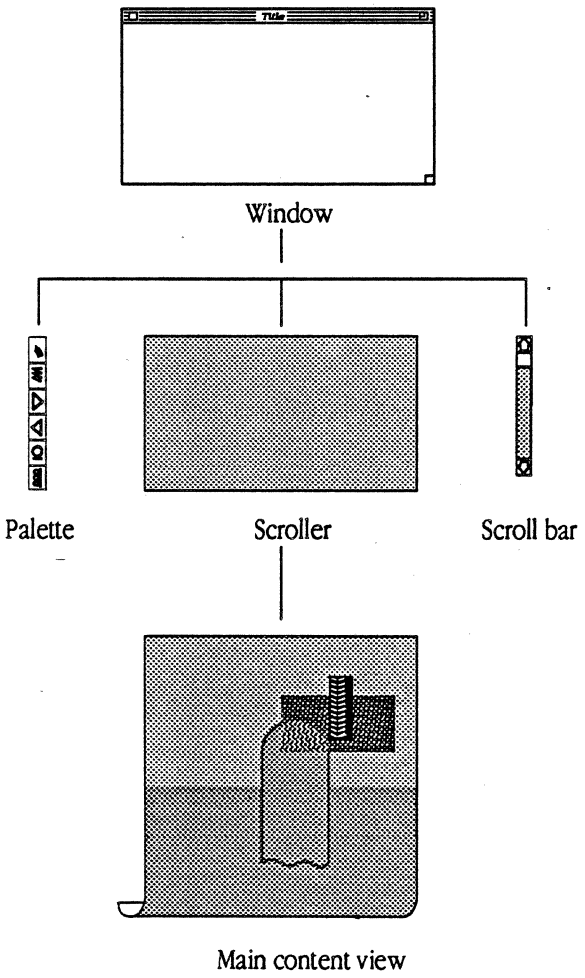
There are instances of five different classes represented in Figure 4-3: a window (TWindow), a scroll bar (TSScrollBar), a scroller (TScroller), a content view (TView or subclass), and a palette (TGridView or TView). Figure 4-4 shows an exploded version of this window.

■ **Figure 4-4** Exploded version of window in Figure 4-3



The view tree for this window is pictured in Figure 4-5. Each of these parts is discussed separately, beginning at the top of the tree, in the next section.

■ **Figure 4-5** View tree for window in Figure 4-3



Windows

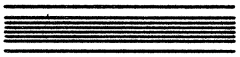
At the top of the tree is an instance of TWindow. TWindow implements standard Macintosh window behavior, and you will need to create subclasses of it only when you want to change its behavior. The superview of every instance of TWindow is NIL, since windows are at the top of the tree of view instances.

The TWindow *class* is an immediate subclass of TView, thus inheriting all the fields and methods of the class TView. TWindow is further customized to give its objects the special attributes of a window; it has fields that record the window's name and the attributes of the window (Can it be closed or resized? How big or small can the user make it? What should happen when the user closes the window?).

TWindow has special methods for opening, closing, activating, and resizing; it also has methods for handling commands a window is likely to receive via the mouse or the menu. For example, if the user clicks the close box, the zoom box, or the size box, TWindow's methods can handle the command. Because the appearance and behavior of windows (but not their contents) are pretty much standardized, MacApp's own code can handle almost anything a window has to do. You don't need to do much with windows, except tell MacApp initially which attributes you want them to have. Note that MacApp does not know how to manipulate the contents of a window, just the window frame and its controls.

- *Note:* If you are familiar with *Inside Macintosh*, you should note that TWindow is MacApp's representation of a Window Manager window. In fact, a Window Manager window is created for every instance of TWindow, and the visual representation of the window depends on the type of Window Manager window you've defined in your resource file.

■ **Figure 4-6** Parts of the window

Name of element	Appearance on screen	How user manipulates control
Close box		User clicks to close the window.
Title bar		User drags to move the window.
Title		Not a control; displays name of window.
Size box		User drags to adjust size of window.
Zoom box		User clicks to make window as large as possible; clicks again to return window to its original size.

The window instance shown in Figure 4-5 has three subviews: a scroller, a scroll bar, and a palette.

Scrollers

The purpose of the scroller class, called TScroller, is to calculate coordinate translations to create the illusion of scrolling through a view, either horizontally, vertically, or both. The user typically accomplishes scrolling by manipulating scroll bars. In that case, the scroll bars and the scroller work in concert. But because MacApp provides a scroller separate from the scroll bars, you may implement any other scrolling mechanism that you care to.

Generally, you can create instances of TScroller directly, without having to override any of its methods, so for the most part you can treat TScroller as a black box. Just be aware that anything scrollable is probably implemented as a subview of an instance of the TScroller class.

Main content

The main content of the window is a separate view, which is often a subview of a scroller, as you can see in Figure 4-5. The content of any window is hard to predict, so MacApp's library of code does not contain specific classes for every possible main content view. You will probably have to create your own subclass of TView that knows how to display whatever it is you need to display. You will probably want to add fields that keep track of what is being displayed, and methods that will do the drawing, probably by calling the Draw methods belonging to the instance of whatever is to be displayed.

Like all other views, the main content view is an event handler. The main content view is usually responsible for a great many of the commands a user is likely to issue. For instance, in a graphics program, if the user is supposed to draw a box by holding down the mouse button and dragging, it is the view instance that will be responsible for intercepting the mouse click and calling the appropriate methods and routines in response.

Palettes

In addition to the window, scroller, scroll bar, and main content views, the particular window in Figure 4-3 also has a **palette**. Since every palette is different, the MacApp library has no special class for creating palettes, so you would probably create a palette class as a direct descendant of TView or TGridView. (Creating a palette is really not much different from creating the main content view.) Palettes are generally not scrollable (though they could be), so usually the palette will have the window instance as its superview. (If it were scrollable, the palette would have a scroller instance as its superview.) The most important methods you need to customize in a palette view are the ones responsible for responding to the user's click inside the palette, especially the methods that highlight and record the selection.

Scroll bars

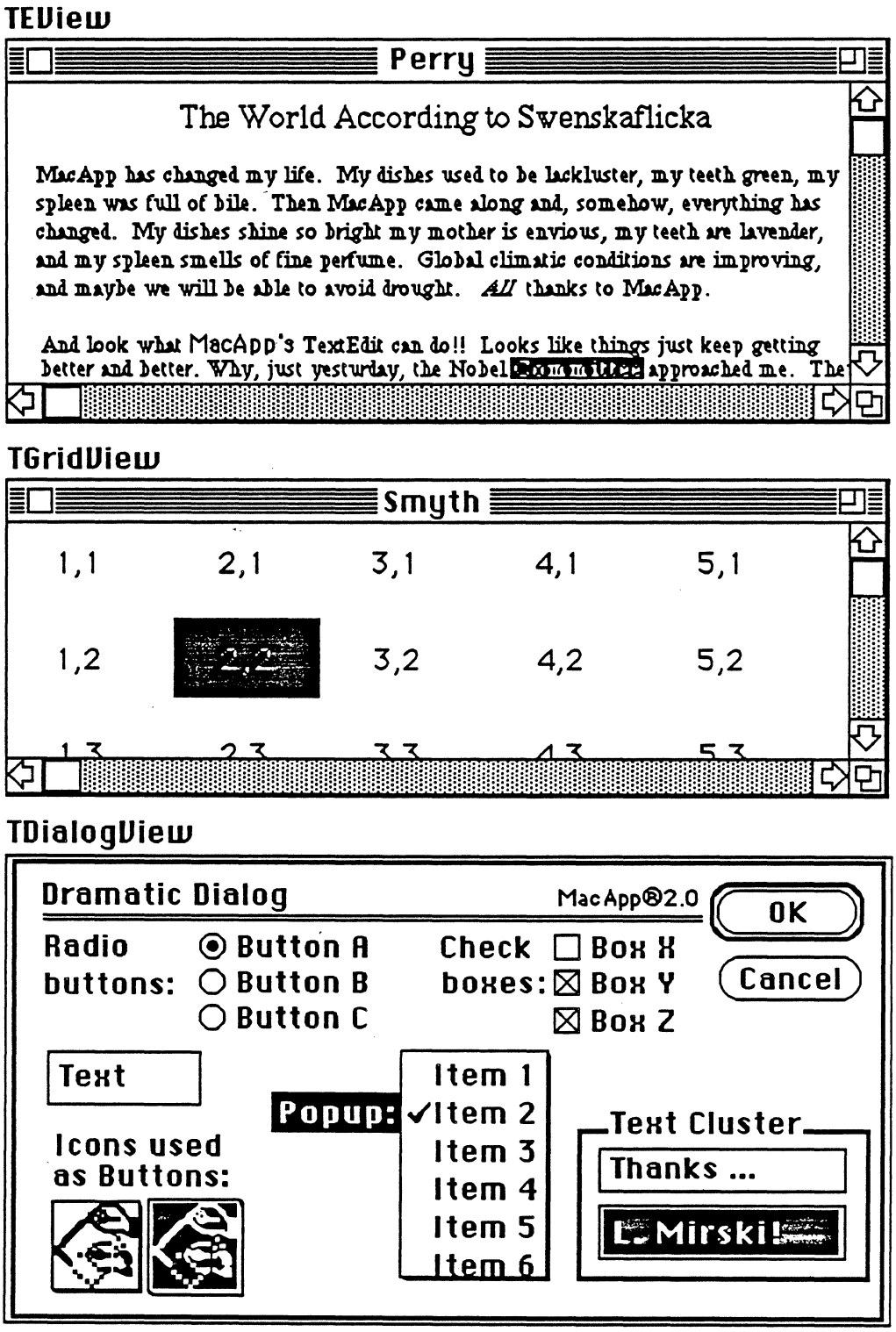
Another object shown in Figure 4-5 is the **scroll bar**. This particular window has only a vertical scroll bar. The scroll bar is an instance of the TSScrollBar class (Ancestry: TObject ← TEvtHandler ← TView ← TControl ← TCtlMgr ← TScrollBar ← TSScrollBar). TSScrollBar is an indirect subclass of TControl, which in turn is a subclass of TView. Because scroll bars are highly conventionalized, you do not need to be too concerned with the inner workings of this class. It has fields that point to the scroller instances it is associated with and that record the limits of scrolling, and it has methods that respond to the user's mouse clicks and maintain the appearance of the scroll bars. Your scroller instance has methods to create instances of TSScrollBar.

- *Note:* MacApp has two classes with similar names, both of which are responsible for displaying and manipulating scroll bars: TScrollBar and TSScrollBar. The latter—the one with two *S*s—is a subclass of the former that has been customized to work with scroller instances; the extra *S* stands for "scroller." The former class, TScrollBar, is a simple control that you can use as an analog dial in dialog boxes.

Other specialized views

MacApp provides several specialized subclasses of TView that you can use to display text, columnar data, or dialog boxes. Text can be displayed and manipulated with an instance of the class TEView, which is part of the optional UTEView building block. Columnar data, such as you might see in a spreadsheet program, can be manipulated and displayed with an instance of the class TGridView. And sophisticated dialog boxes, including static text, editable text, check boxes, radio buttons, and so on can be manipulated and displayed with a wide variety of classes found mostly in the UDialog unit. These specialized views are pictured in Figure 4-7. (Appendix A, "Changes Since MacApp 1.1," lists all the view classes provided in MacApp 2.0.)

■ **Figure 4-7** TEView, TGridView, and TDialogView



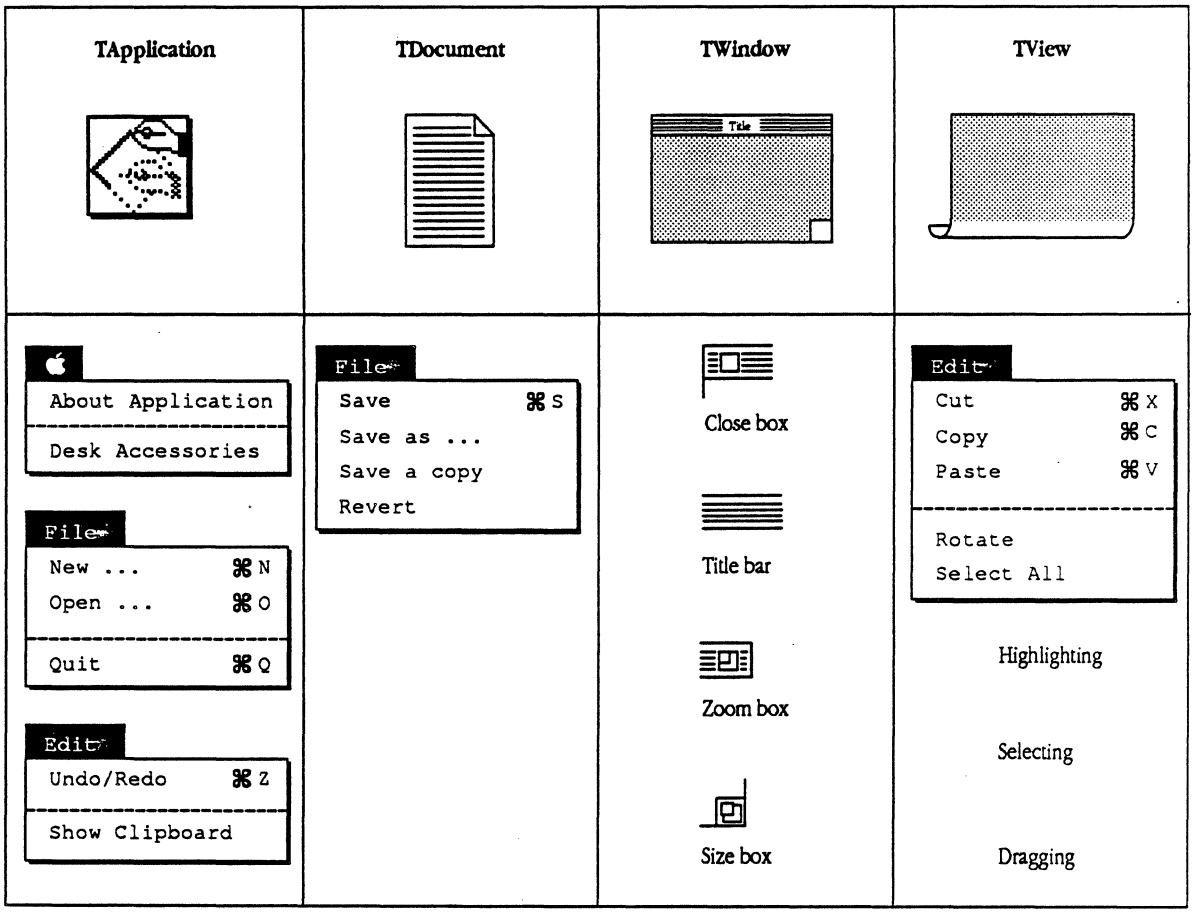
Executing commands

In addition to storing and displaying data, applications must allow users to manipulate the data. In the Macintosh world, any input from the user, even typing simple text, is considered to be a command. However, because MacApp reserves the word *command* for menu commands, the term **user event** is used to describe many kinds of commands. Users have two tools for creating user events: the mouse and the keyboard. With the mouse, the user can choose commands from menus or click on a window's controls or its content. In the content of a window, users can click an object to select it, drag an object to move it, or click in a palette to choose a tool. With the keyboard, users can type text or can enter Command-key combinations. Command-key combinations are equivalent to menu items; for instance, there is no difference in most applications between choosing Save from the File menu and typing Command-S from the keyboard.

When the user chooses a menu command using either the keyboard or the mouse, MacApp receives a report of the event from the Macintosh System. Methods in the MacApp library begin a search for the correct instance to handle each sort of event. (The algorithm for this search is the topic of the next chapter.) But there are rules of thumb for figuring out which instances are responsible for handling which commands. First of all, only event-handler instances respond to user events. The main players in the game are the application, document, window, and specialized view instances. MacApp gives each of these objects a shot at resolving any incoming command, in a specific order. There is generally only one candidate for each class. For example, only one view, window, and document are active at any one time, and there can only be a single instance of the TApplication class.

Each class is responsible for handling user events related to its own area. The application object is responsible for user events that could be executed even when no window is open, including Quit, About, New, and Show Clipboard. The document is responsible for reading and writing data to disk. The window is responsible for clicks in the size box, title bar, and zoom box. Scroll bar instances handle clicks in the scroll bars. The view is responsible for manipulating the images it shows; usually it has methods that actually take input from the mouse to draw or manipulate an object, and then report the object's existence to the document so it can be recorded.

■ **Figure 4-8** The main event handlers and their responsibilities for commands



The most important methods for handling the user's commands are DoMenuCommand, DoMouseCommand, and DoKeyCommand. Because these methods are declared in TEvtHandler, they are shared by all event handlers. When referring to them as a group in the following discussion, we will call them the Do*Whatever*Commands.

The Do*Whatever*Commands split up responsibility for the various user events as follows:

- DoMenuCommand handles all menu commands, whether they come from a mouse click or from a Command-key combination.
- DoMouseCommand handles mouse clicks that occur outside the menus.
- DoKeyCommand handles all keystrokes *not* combined with the Command key.

These three methods generally do *not* do the work of the command itself—they either call other methods of the same instance or methods of some other instance. The particular manner in which the Do*Whatever*Command does its work depends on whether the user chooses a simple command or a complex one. A command is simple if it is not undoable and does not require the mouse to be tracked; a complex command is undoable or requires mouse tracking. Most Macintosh commands are complex because they change the document in one way or another, and should therefore be undoable, or because they require the mouse to be tracked.

■ **Table 4-3** Examples of simple and complex commands

Command	Reason
Simple commands	
Save (menu command)	Cannot be undone and does not require the mouse to be tracked.
Select All (menu command)	Cannot be undone and does not require the mouse to be tracked.
Complex commands	
Delete or Clear	Must be undoable since it changes the document.
Select item (with mouse)	Requires mouse to be tracked.
Move item (with mouse)	Requires mouse to be tracked and must be undoable.

The Do*Whatever*Command method usually executes simple commands by calling other methods. For example, the Save command is handled by a document instance that calls the document's own Save method. The Do*Whatever*Command executes complex commands by creating an instance of some command class, whose methods will then be called by MacApp. For every complex command you wish to implement, you will need to design a command class that is a subclass of TCommand.

Remember that the TCommand class is an abstract class. It has fields that record important information about the command, including a reference number of the command, the relevant view instance, the object instance that is the target of the command, and information about the status of the command, such as whether the command is undoable, requires tracking, and whether it caused a change. It has methods that track the mouse and that do and undo the command.

In sum, simple commands are executed without creating new objects, but complex commands—undoable commands or commands that require mouse tracking—require that an object that is a member of a command class be created.

Summary

Now you should understand how MacApp is organized: how the files and units are related; the significance of the ancestry tree for the classes; and, most important, the conceptual design behind the MacApp library. The next chapter moves from this static view of organization to a dynamic picture of how a MacApp application works while it is running, focusing on how it handles events.

Chapter 5 Introduction to MacApp: Flow of Control

PEOPLE LEARNING MACAPP OFTEN ASK THEMSELVES, "Just who is in charge here, my code, or MacApp's?" At any moment, the user might click the mouse button; some part of the MacApp library would handle the click by calling a piece of your code. Your code would probably then return control to MacApp, which in turn might call a ROM routine to draw something on the screen. Then the user might click something else, starting the process over again. Thus, control over the computer is constantly traded among the user, your code, MacApp, and the ROM routines. This perpetual shuffling is called **flow of control** and is the subject of this chapter. ■

Event handling in MacApp

Any application might have to handle a great number of events while it is running; fortunately, they fall into a very few categories. Macintosh applications spend most of their time idling in an outer loop, called the **main event loop,** waiting for something to happen. When an event comes along that needs to be handled, the main event loop sorts it into one of these categories, opening a path down which the event will flow until it is taken care of. If you find it hard to picture this, imagine one of those coin banks that sorts your change into piles of pennies, nickels, dimes, and quarters. Every time a user creates an event, it is as if he or she has dropped a coin into the top. MacApp, like the coin bank, evaluates the event and sends it down the correct chute.

In MacApp there are eight chutes or categories, each representing a different sort of event. Each event falls into one of these eight categories. After the event starts down its path, MacApp must decide which instance should handle it. The process of assigning events to instances is one of the more subtle aspects of flow of control in MacApp, and it is the purpose of this chapter to help you understand how those decisions are made. Some of these assignments will be intuitive to you, even from the beginning; others will not concern you, because some events can be handled without ever having to call your customized code; but still others use wily algorithms to do the assigning, algorithms you must understand to create the proper code to handle them. Chances are, once you understand these, you will understand MacApp.

This chapter begins by explaining the main event loop and the mechanism by which events are assigned to instances. The chapter then discusses which of the eight categories ought to concern you, and how you can get a feel for which events will be assigned to which instances.

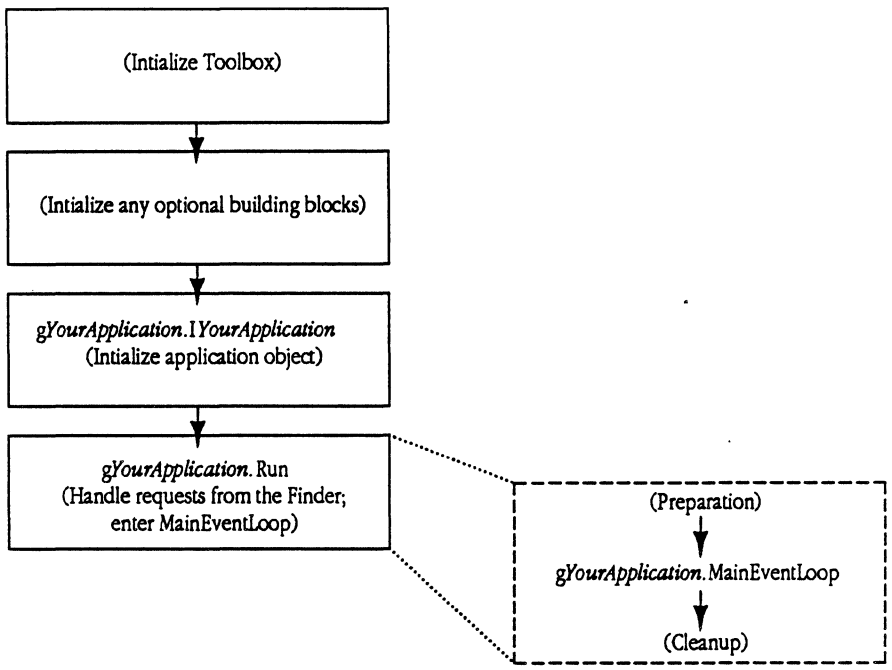
Overview of MacApp

In the beginning, you may be curious about how MacApp works, and there is nothing to stop you from opening up MacApp's code and reading along. Reading the raw code can be frustrating—it is hard to tell what's a method, what's a global routine, and what comes from ROM. Moreover, you can get lost in the detail of the code. The flowcharts in this chapter allow you to see an overview, so if you do decide to read the source code, at least you'll have a road map.

Once the program is running, your application will spend all its time in the methods of your various instances, except for a few lines of startup code and some global routines here and there which you or MacApp will call. A simple mouse click in one of your application's windows could generate messages among a handful of objects and several dozen of their methods. Rather than presenting you with every possible path along which the data could flow for a particular event, the flowcharts present the flow for some specific event, which will be explicitly defined. In order to keep most of these flowcharts small enough to fit on one page, some parts will be abstract and not detailed.

The first flowchart, in Figure 5-1, is an overview of an entire MacApp application, from the time the user double-clicks the icon to the time he or she quits. It is not very detailed or very interesting, and doesn't look much different from a traditional procedure-based program. (Also, it is a good example of a limited chart. There are two other paths an application could take when it starts up, and this one shows only the simplest path—it assumes the user opens the application from the Finder™ without requesting that any documents be opened or printed.)

■ **Figure 5-1** Fundamental overview of a MacApp application

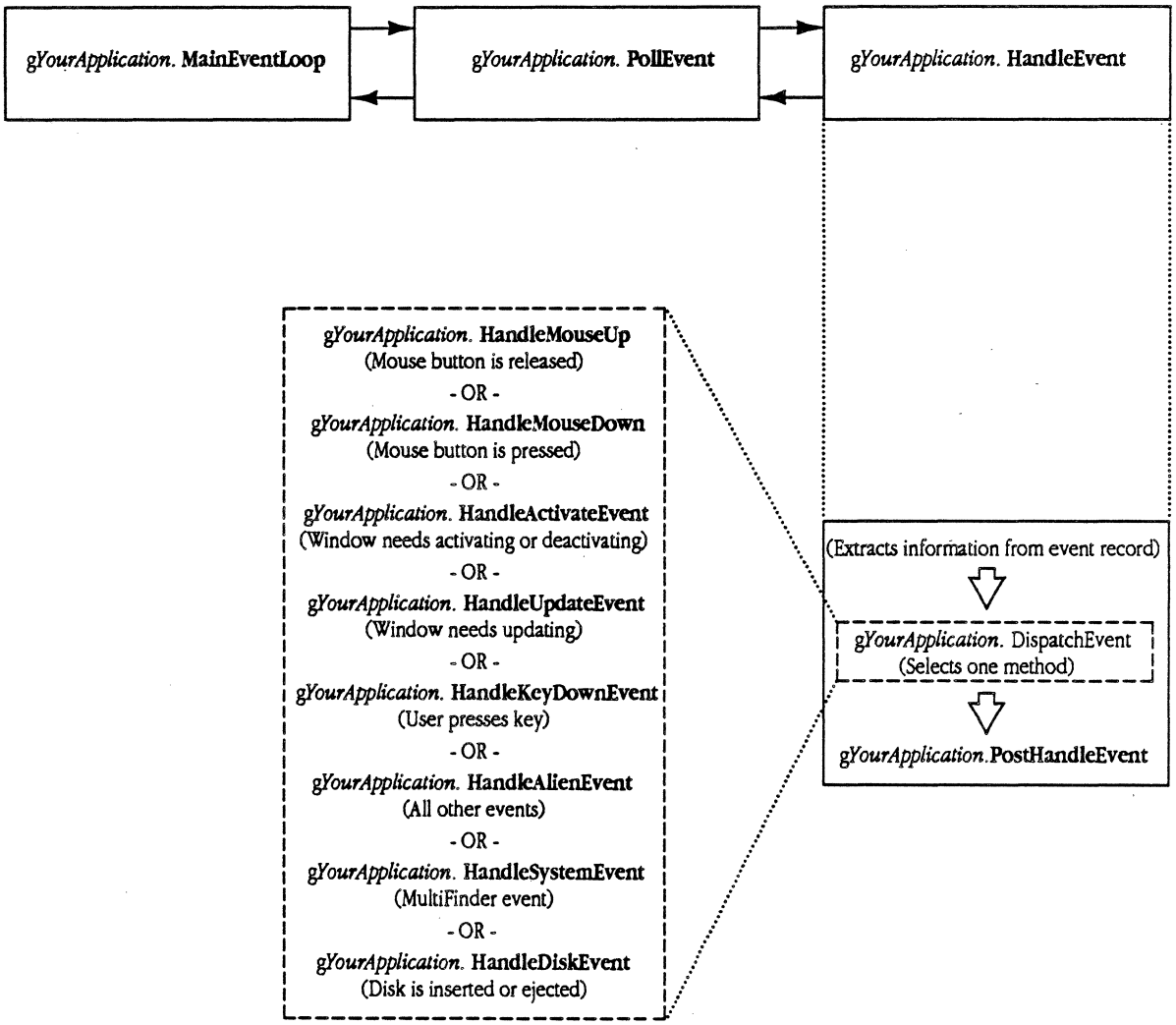


One of the methods, MainEventLoop, is the loop discussed previously. Your program will spend much of its time in this loop. Much of the rest of this section will be spent investigating this method of the TApplication class; even so, only the first few layers of its complexity will be described. In this chart, the flow is entirely linear, starting at the top and dropping off the bottom, which is the end of the application. Notice that the routine Cleanup comes at the end of the program; it takes care of last-minute details. Other routines, though they require further explanation, are not very important for helping you to grasp the basic idea of MacApp.

Figure 5-2 is the first nonlinear flowchart. The action both starts and ends at the method MainEventLoop. The first thing this flowchart tells you is that MainEventLoop calls PollEvent, which returns control to MainEventLoop after it's finished. MainEventLoop repeatedly calls PollEvent until the user chooses Quit.

But the flowchart also tells you that if an event is ready to be handled (or, more accurately, if a non-null event has occurred), PollEvent calls HandleEvent. Furthermore, it tells you what happens inside HandleEvent: The record of the event, which comes from ROM, is stored in a special data configuration. Afterward, HandleEvent calls DispatchEvent and PostHandleEvent.

■ **Figure 5-2** Flow of control: MainEventLoop



Finally, the flowchart details how the DispatchEvent method works. The *OR*'s between the boxes indicate that only one of these methods is called; the condition for choosing that particular method is indicated in parentheses in each box. In fact, as you might have guessed, this all comes in the form of a CASE statement in the code. (See Figure 5-3.)

Finally, the flowchart details how the DispatchEvent method works. The *OR*s between the boxes indicate that only one of these methods is called; the condition for choosing that particular method is indicated in parentheses in each box. In fact, as you might have guessed, this all comes in the form of a CASE statement in the code. (See Figure 5-3.)

■ **Figure 5-3** The CASE statement in TApplication

```
CASE what OF
      mouseUp:
      commandToPerform := HandleMouseUp(theEventInfo);

      mouseDown:
      commandToPerform := HandleMouseDown(theEventInfo);

      activateEvt:
      commandToPerform :=
            HandleActivateEvent(theEventInfo);

      updateEvt:
      commandToPerform :=
            HandleUpdateEvent(theEventInfo);

      keyDown, autoKey:
      commandToPerform :=
            HandleKeyDownEvent(theEventInfo);

      diskEvt:
      commandToPerform := HandleDiskEvent(theEventInfo);

      app4Evt:
      commandToPerform :=
            HandleSystemEvent(theEventInfo);

      OTHERWISE
            commandToPerform :=
                  HandleAlienEvent(theEventInfo);

      END; {case}
```

The overall picture should now be apparent. MainEventLoop repeatedly calls PollEvent. Whenever there *is* an event, PollEvent will call HandleEvent, which will record it, dispatch it by calling one of nine methods, and then execute PostHandleEvent. When PostHandleEvent completes execution, HandleEvent also finishes and returns control to PollEvent, which in turn returns control to MainEventLoop. The process repeats until the user chooses Quit. Keep in mind that you can override any of the methods listed in this chart if you want to change the way MacApp handles a specific type of event. However, MacApp is capable of handling most events the user can generate; but there are other events that it does not anticipate; these are called **alien events.** You must override HandleAlienEvent if you wish your application to respond to these events.

Note that the flowchart in Figure 5-2, like all the others in this book, is incomplete in several ways. First, it does not tell you what happens in PollEvent if there is not an event waiting. Second, it does not tell you everything that happens. For instance, it has left out all the error-checking code and the method call that actually checked for the event. This streamlines the flowchart, allowing you to focus on the main points.

Assigning events to instances

By now, you should have a mental picture depicting MacApp constantly on the lookout for events. If PollEvent sees an event, it will pass it on to DispatchEvent, which will send it down one of the eight possible paths. (If there is *no* event, PollEvent will open a ninth path so any work that should be done while the application is idling can have an opportunity to execute; there will be more about idling later in this chapter.) Now it is time to look at these eight paths in more detail.

First, how does MacApp know which sort of event has taken place? This is perhaps the easiest question of all. Whenever there is an event, it is intercepted by Toolbox routines that report exactly what sort of event took place. All DispatchEvent has to do is look in a certain field of the event record. What it will find there is one of the following eight sorts of events:

MouseDown	The user presses the mouse button. What action MacApp takes depends on where the cursor was when the button was pressed and where the mouse was dragged while the button remained down.
MouseUp	The user releases the mouse button. Generally this event is ignored, except when the user has dragged the mouse.
Activate	A window is activated or deactivated. In general, an activate event occurs as an indirect result of a user action. For instance, a user may click on an inactive window, which in turn will cause the Window Manager to generate two activate events, one to deactivate the old window and another to activate the new one.
Update	A window needs refreshing. Usually an update event occurs as an indirect result of a user action. For instance, if a user had one window partially covered and then uncovered it, the Window Manager would generate an update event for the uncovered region to be redrawn.
KeyDown	The user presses a key or a key combination on the keyboard. KeyDown events include both plain keystrokes and Command-key sequences (a key pressed while the Command key is held down).
Disk	The user inserts or ejects a disk.
System	The user enters or leaves the application with MultiFinder or Switcher, or Multifinder generated a "mouse moved" event.
Alien	Other events, such as network events or events that you define yourself, are called alien events. By default, MacApp ignores alien events.

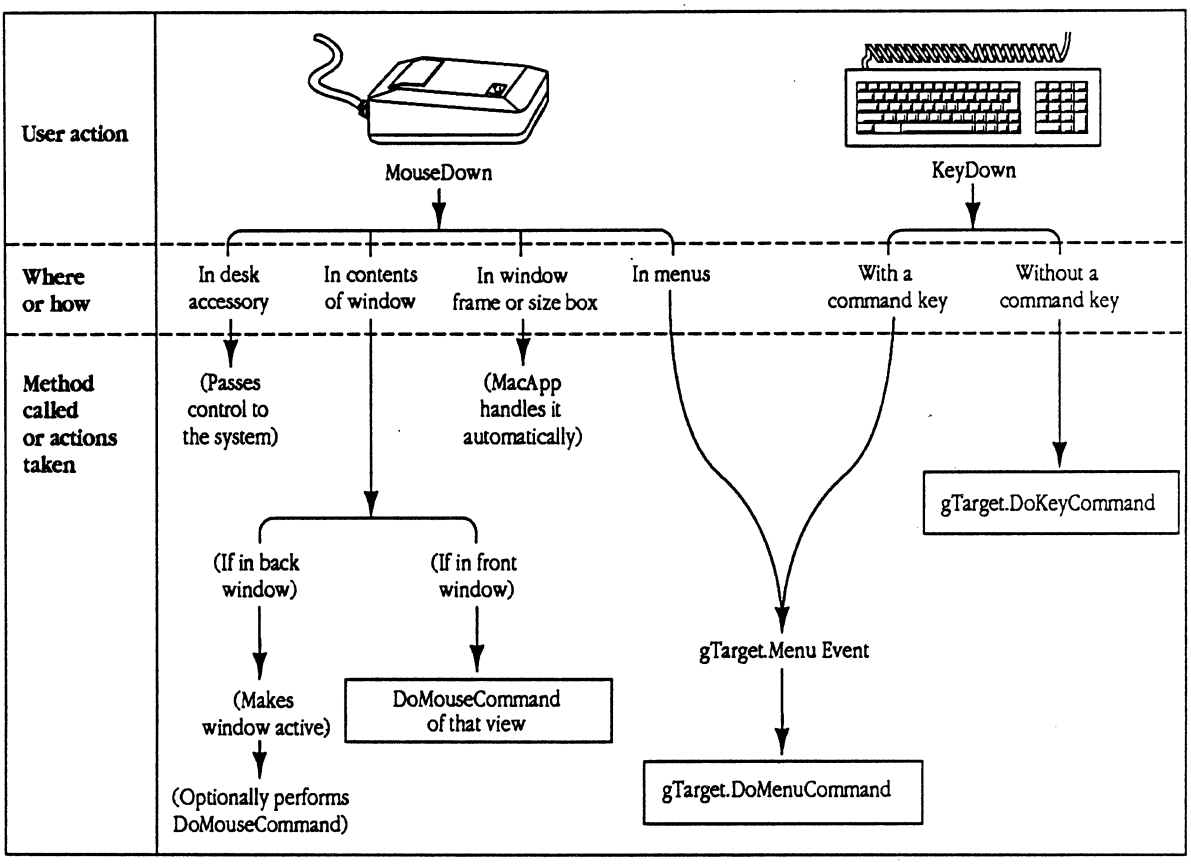
Which of these events do you need to be concerned about? MacApp's code can take care of many of these events for you, as long as you are happy with the default operation MacApp gives you. In fact, that's a crucial idea behind MacApp. The application instance catches events as they come in and assigns them to the instances responsible for handling them. *You* don't have to be concerned with events themselves. Instead, all you need to know is which methods and classes you have to implement or override. But in order to give you some intuition about which methods and classes are of concern, here is an overview of how MacApp deals with events of various sorts.

Three of the classes of events are dead ends that you can typically ignore: MouseUp events are generally discarded or handled automatically; Disk events require no action except error checking; and Alien events can be neglected, unless you are writing a networking or special-purpose program.

- *Note:* There are three typical cases when MouseUp events are not ignored: when the user drags the mouse; when the user double- or triple-clicks the mouse; and when the user releases the mouse button over a TControl instance. In each of these cases, MacApp's code intercepts the event and handles it appropriately.

Three more of the event classes require consistently predictable actions, which MacApp can take care of for you, so you can ignore them unless you want some special action: Activate, Update, and System events require no special attention, except that update events will call the Draw methods of any views that need updating. Since MacApp cannot anticipate how you want views to be drawn, TView's default Draw methods are empty. You must override them if you want anything to be drawn. (Note, however, that many of TView's subclasses do have functional Draw methods.) All these considerations aside, that leaves two sorts of events you need to pay special attention to: KeyDown and MouseDown events. These events can be further subdivided as shown in Figure 5-4.

■ **Figure 5-4** The events most important to you



As we see in Figure 5-4, there are a variety of ways mouse clicks can be handled, depending on the context of the click. MacApp learns about this from information provided by the ROM in the event record. By default, the MacApp code will handle many of the MouseDown clicks by itself, including clicks in the various window controls, such as the scroll bars and the zoom, size, and close boxes. MacApp will also handle clicks in any desk accessories by handing control over to the system. The two MouseDown events you should be concerned about are clicks in the content of a window and clicks in the menus. These may require customized action from your application, action MacApp cannot anticipate. Figure 5-4 also shows two sorts of KeyDown events: ordinary keystrokes and Command-key combinations, both of which may require your customized code.

Because Command-key combinations are treated as equivalents of menu commands, there are only three sorts of MouseDown and KeyDown events that your code absolutely must handle: (1) clicks in the content of a window, (2) plain keystrokes, and (3) menu commands (or their keyboard equivalents). There is a fourth area that may be of concern to you: What happens if there is no event waiting? In that case, the program enters an Idle state, treating the idling as if it too were an event (instead of a nonevent), optionally executing your customized code that may, for instance, maintain cursors or check for network activity. By default, when your application is idling, the application instance calls methods that can change the shape of the cursor depending on where the user has moved it, but even these methods are empty until you override them.

So these are the four truly interesting sorts of events—these are the places, in addition to the Draw methods of the TView instances, where much of your customized code will come in. How do you write code that MacApp will call? MacApp calls methods with certain names in certain circumstances. When MacApp responds to a click inside a window, it calls a method called DoMouseCommand. When MacApp responds to a menu command (or Command-key equivalent) it calls a method named DoMenuCommand; and a plain keystroke is handled with a DoKeyCommand method. Finally, when MacApp is idling, it calls DoIdle methods. You must override these methods to specify how these events will be handled.

But methods belong to particular instances. How does MacApp's code determine which instance should handle each event? All the events are handled by some subclass of TEvtHandler, but that doesn't narrow it down much: Should a command be handled by a view, a window, a document, or the application instance? MacApp follows a set of protocols that invoke one of several chains of responsibility to determine which object is responsible for handling a particular event. Conceptually, there are three of these chains: The command chain takes care of keystrokes and menu commands (whether they are chosen by clicking menu items or typing Command-key combinations); The click chain takes care of mouse clicks in the content region of a window; and the idle chain takes care of idle "events."

Target chains

A chain is a series of connected or related things. In MacApp, a **target** is the object instance upon which some command will act. Just as a common metal chain is made up of individual links joining those at either end, a target chain in MacApp is a linked list of instances, with a beginning and an end, joined by **object reference fields**. In response to a user event, MacApp traverses the target chain from that event's source to its destination: the event handler that executes a command in response to the user's action.

It is helpful to think of the target chain in terms of the event handlers in it; thus, we shall discuss the **command chain** , the **click chain** and the **cohandler chain** as if they were separate entities. However, you should be aware that at any given moment there is only one target and therefore only one target chain. As MacApp traverses the chain, one link at a time, it checks to see if each object along the way can handle the event. Once an object handles the event, the search may be called off.

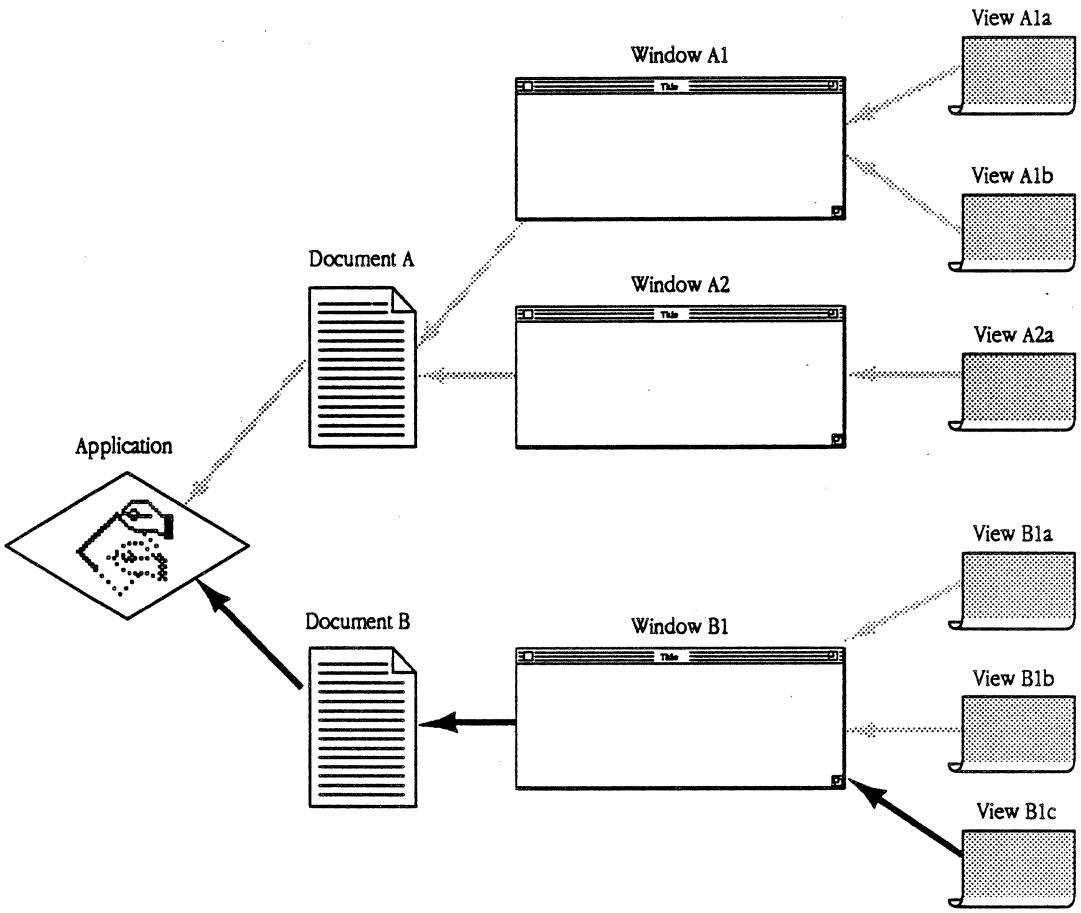
Because there is not necessarily a linear, chain-like relationship among the instances in a MacApp application, the method by which this chain is constructed from some underlying structure is a matter of importance, as is the method by which the head of the chain is chosen and the method by which the chain is traversed.

The command chain

The **command chain** is traversed whenever the user types at the keyboard or chooses an item from a menu, whether the item is chosen by pointing and clicking or by typing its keyboard equivalent. The command chain can be complicated because commands may be handled by almost any subclass of TEvtHandler, and it is not obvious where the search should start or how it should proceed. Certain commands should belong to a document, such as commands that save or print a particular document; others belong to a window, such as a zoom command; still others might belong to a particular view, such as the Cut and Paste commands. Still others might belong to the application as a whole, if they should be available all the time and do not concern a particular document, window, or view—for example, the New or About commands are always available even if no window is open.

But these instances do not at all have a linear relationship. Instead, they form a **tree**, with the single application instance at the root and views and subviews as the branches.

■ **Figure 5-5** Target chain from View B1c to the application object



The different instances keep track of each other by way of object reference fields, represented by the arrows in the figure. So views have reference fields that point to their superviews. Windows have fields that point to documents. And documents have access to the variable that points to the application.

How the chain is built

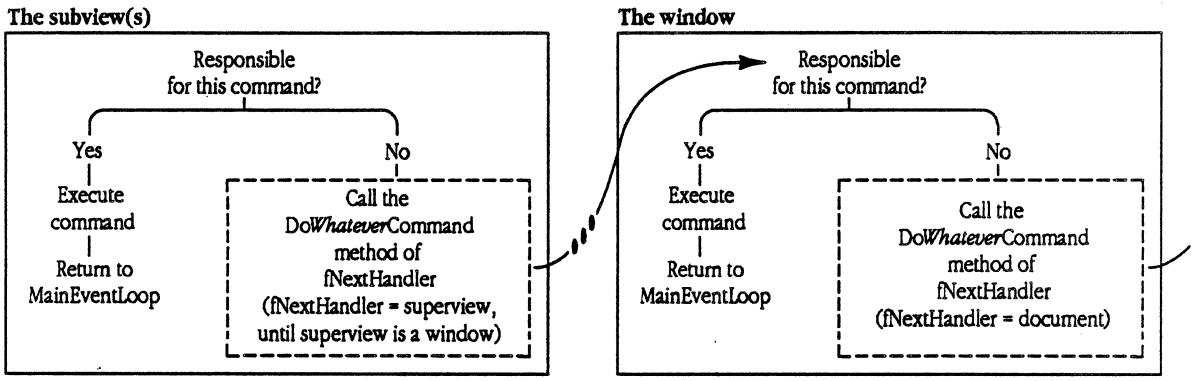
Every subclass of TEvtHandler has a field called fNextHandler, which stores the next link in the chain. When an event-handler object is created, its fNextHandler must be given an initial value that will reference the appropriate target for events which that object cannot handle. MacApp takes care of the initialization of windows and most views and subviews, initializing each event-handling instance to have the correct fNextHandler. (You may, however, have to provide additional initialization methods for your own specialized views or command objects.) Subviews get initialized so that their fNextHandler points to their superview, which may or may not be a window. But since every subview is inside some window (which is itself a view object), this chain will eventually point to a window object.

Window objects are initialized with their fNextHandler field pointing to the document they are primarily associated with. (When you create the window, you provide this information by passing the document's handle as a parameter.) Documents in turn are initialized so their fNextHandlers point to the application. Again, MacApp can do a certain amount of this for you, but you will need to provide any code you need to set up an initial state for your specialized document; for instance, you might want to insert initial values in a spreadsheet template, or set the initial state of some controls when opening a new document, such as the default margins in a new word processing document.

Some windows, like palette windows, are not associated with any document; these windows point directly to the application. The application points to NIL, so it is the end of the chain. If you look at the four objects connected by the bolded arrows in Figure 5-5, you can see one target chain.

The net result is a sensible chain of responsibility from specific to general. A subview gets first crack at any command or keystroke, followed by the window, the document, and finally the application. You may modify the chain by changing the fNextHandler field of any object.

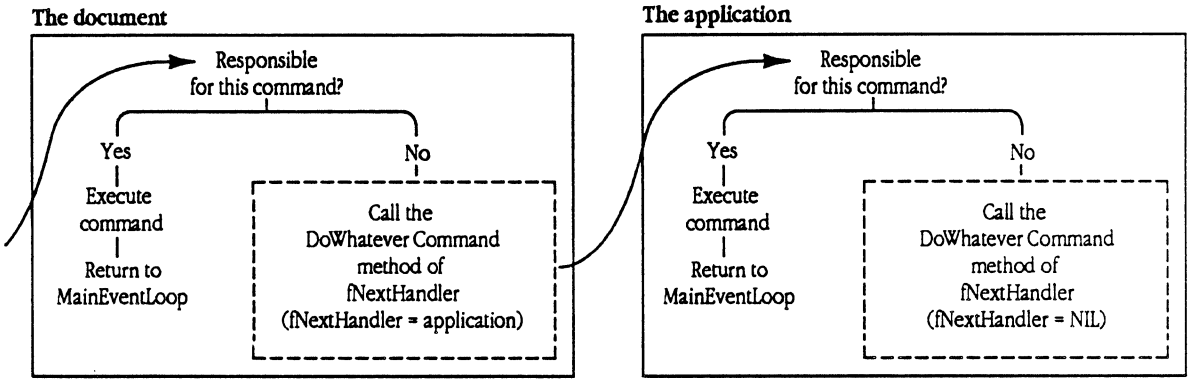
■ **Figure 5-6** Diagram of fNextHandlers



How the head of the chain is determined

MacApp has a global variable named **gTarget** which always points to the head of the chain. The philosophy behind gTarget is that there is always some part of the application where the user fixes his or her interest, and that is where gTarget should point. There are only a few methods in MacApp that set gTarget, the most important of which is TWindow.Activate, a method belonging to window instances. This method is called whenever a window is activated or deactivated. When you, as the programmer, write the methods that create the window objects, you will specify which other instance should become the target when that window is activated. This will almost always be the most important or topmost view inside the window. That information is stored in a field of the window instance, a field that TWindow.Activate checks whenever the window becomes active.

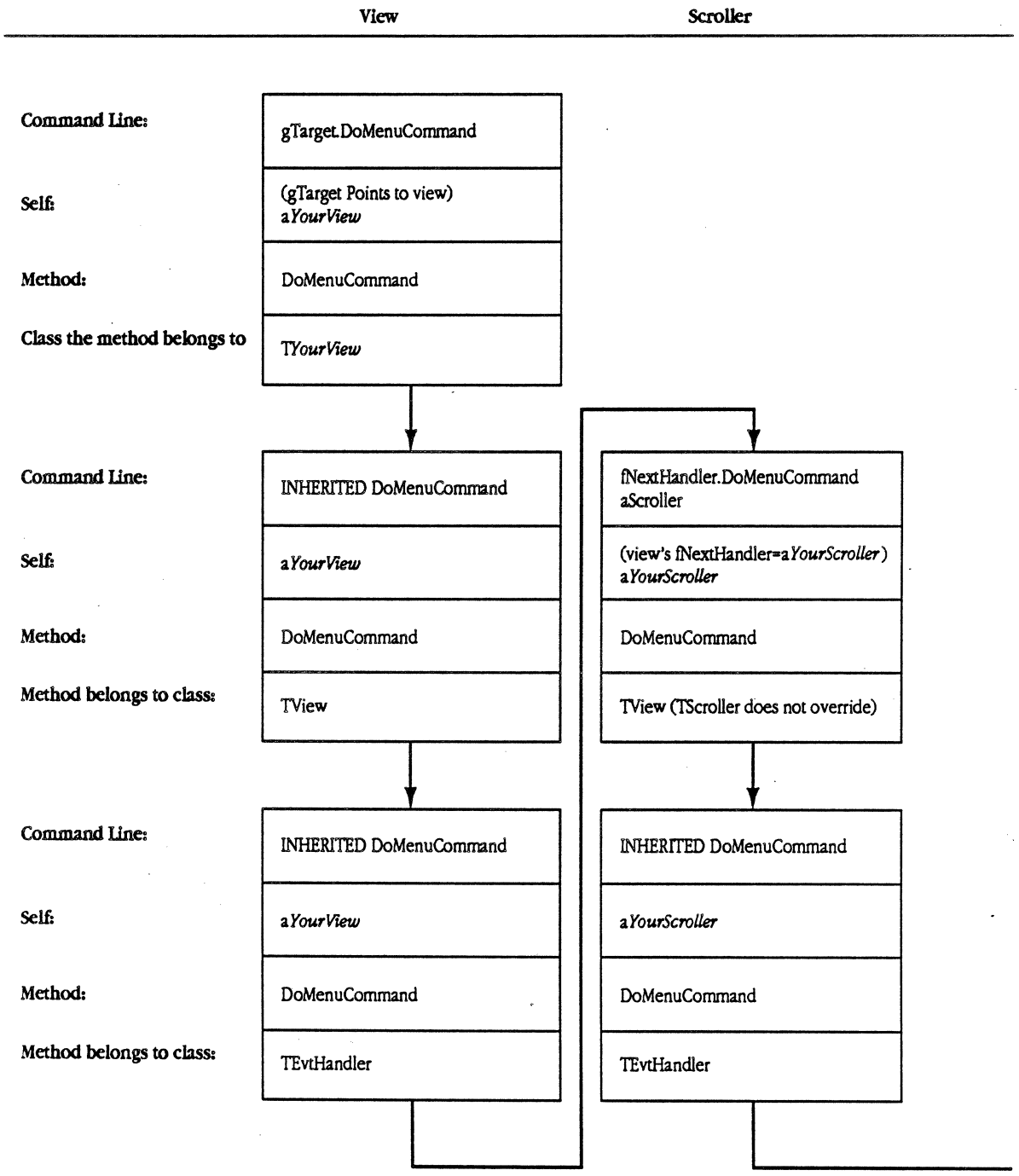
■ **Figure 5-6** Diagram of fNextHandlers (continued)



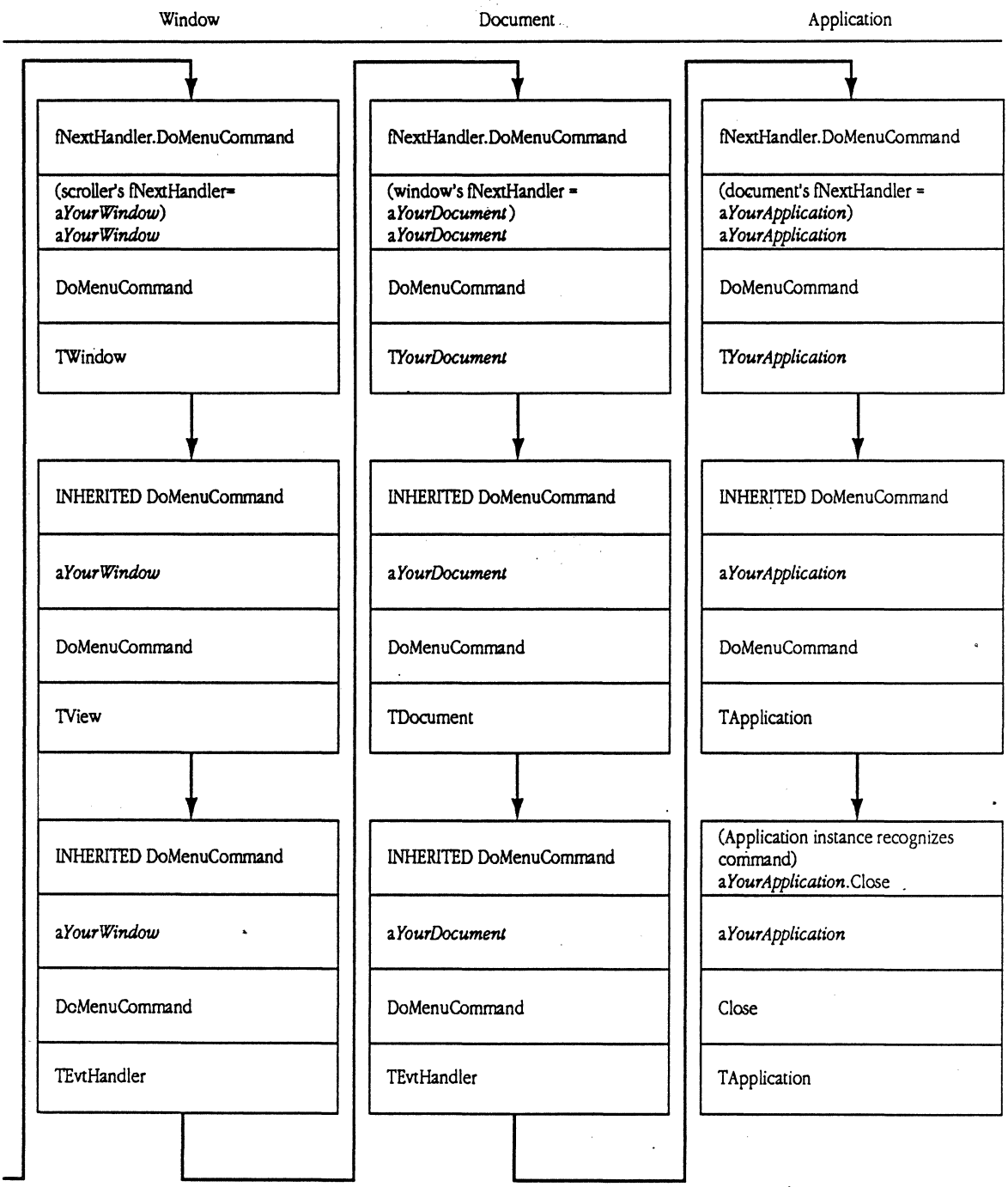
How the chain is traversed

When the user types a plain keystroke or chooses a menu command, MacApp calls a method of whatever is currently gTarget. In the case of keystrokes, it calls gTarget.DoKeyCommand; in the case of menu commands (or Command-key equivalents) it calls gTarget.DoMenuCommand. These methods are often CASE statements that call the inherited DoMenuCommand or DoKeyCommand if none of the specified conditions is met. Eventually, this inherited method will go through the class hierarchy and reach TEvtHandler, which will call a DoMenuCommand or DoKeyCommand belonging to whatever the fNextHandler instance is. This process continues on down the chain until one of the links handles the event or has an fNextHandler equal to NIL. (The application instance always has an fNextHandler that equals NIL. See Figure 5-6.)

■ **Figure 5-7** Possible command chain when the user chooses Quit



■ **Figure 5-7** Possible command chain when the user chooses Quit (continued)



For example, imagine that a user is in a program that was written with MacApp. Imagine that when the user chooses the Quit command, there are four instances: a main content view called yourView; a window called yourWindow; a document called yourDocument; and an application called yourApplication. If the user were inside a window, gTarget would probably point to the view instance. Then, if the user chose Quit, MacApp would first call gTarget.DoMenuCommand. Remember that gTarget points to the item most likely to be of interest. When a window is activated, that is usually one of its views. Since the user was inside a window when he or she issued the command, gTarget would point to the view and yourView.DoMenuCommand would be called. Since Quit belongs to the application object, the view's method would not handle the event. (See Figure 57.)

Remember that view instances have an fNextHandler that points to the superview, in this case a window. Consequently, the next link in the chain would be yourWindow.DoMenuCommand, which also would not handle the event, and gTarget will then go to the method of the fNextHandler. Next in the chain would be yourDocument.DoMenuCommand, which would not handle the event, but instead pass it to yourApplication.DoMenuCommand, which was preprogrammed to know about the Quit command (since all Macintosh applications are assumed to have Quit commands).

The click chain

The **click chain** is responsible for handling mouse clicks in the content region of the window. That means it handles all window clicks that aren't in the controls of the window, including palette clicks and clicks that select and drag objects. The click chain is much simpler than the command chain, partly because window clicks are always, by definition, handled by view objects, and the relationship of view objects to each other is simpler than the relationship between all the possible command-handling objects. View objects form a simple tree, with the window as the root, and views and subviews as the branches and leaves. (See Figure 5-8.)

How the chain is built

Every time you add a subview to a particular view object, MacApp places a reference to that subview in a list kept as a field of that object.

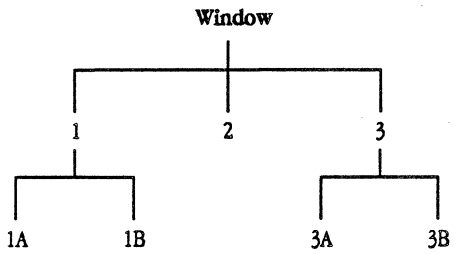
How the head of the chain is determined

Whenever a view is clicked, MacApp invokes a clever recursive routine, HandleMouseDown, that goes up each branch, looking until it finds the particular subview that contained the click. That view is treated as the head of the chain.

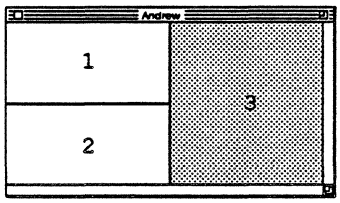
How the chain is traversed

MacApp expects each subview to handle the clicks in its own boundaries, so the event dies after the very first link in the chain. However, if you want to, you can write custom code that causes the event to work its way back up the chain by calling the appropriate method from this view's superview.

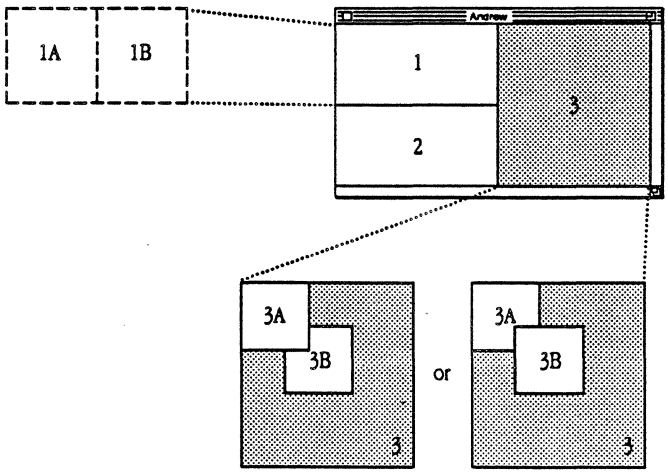
■ **Figure 5-8** The view tree



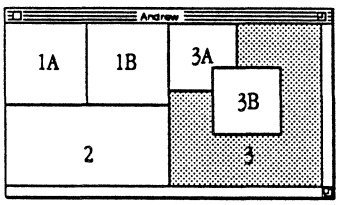
Window has three subviews



Views 1 and 3 each have 2 subviews.



Window and its subviews as the user sees them.

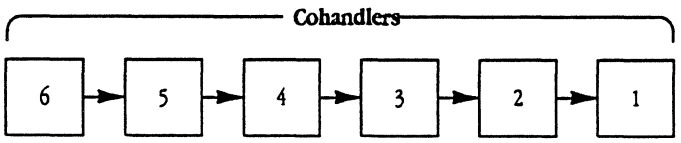


The cohandler chain

While it is hard to imagine writing any useful application without using both the command chain and the click chain, you could easily write many marketable applications knowing nothing about the **cohandler chain**. The cohandler chain actually pulls double duty, providing a chain of responsibility for both idle-time events and alien events. This may seem like an odd combination, but it happens that many objects that handle alien events may also need to take advantage of idle time. In any case, if you will not be encountering any alien events and choose not to do anything fancy with the application's copious idle time, you can avoid the cohandler chain altogether. The cohandler chain is also sometimes referred to as the **idle chain** precisely because it is the place in MacApp where idle events are handled.

Idle objects, including those that handle alien events, bear no underlying relation to each other, so the order of the chain is arbitrary; MacApp links the idle objects together in a linked list, similar to the target chain discussed earlier. When MacApp traverses the cohandler chain because of an Idle event, it calls the DoIdle method; when it traverses the chain because of an alien event, it searches for a DoHandleEvent method.

■ **Figure 5-9** The cohandler chain



How the chain is built

You call the TApplication.InstallCohandler method for each idle object instance you create. InstallCohandler simply links each piece of the chain to the previous piece.

Each idle instance has a field called fIdleFreq that you set to indicate how often its DoIdle method should be executed.

How the head of the chain is determined

MacApp sets the most recent link to be the head of the chain and the oldest link to be the end of the chain. A reference to the head of the chain is kept in a global variable called gHeadCohandler.

How the chain is traversed

The chain will be traversed either when it is time to idle or when an alien event is received. In both cases, the system of traversing the chain is similar to the method for traversing the command chain. In fact, unless you override MacApp's code, every time MacApp traverses the idle chain, it first traverses the command chain looking for DoIdle or DoHandleEvent methods. So the only time you need to install an object in the idle chain is when you want its DoIdle method to be executed even when it is *not* in the command chain. For instance, imagine you had a view with a flashing border around it. If you wanted the border to flash only when the view (or one of its subviews) was active, you would *not* have to place it in the idle chain. If you wanted it to flash no matter what, you *would* have to place it in the chain.

How often your application idles is determined by the fIdleFreq field of each instance in the idle and command chains. When there are no events pending, MacApp will check the fIdleFreq field of each instance in the idle chain. If enough time (measures in **ticks**) has elapsed since the object was last idled, then its DoIdle method is called. DoIdle is overridden for each type of object that requires processing at idle time. (For example, TTEView overrides DoIdle to make the cursor bar flash at the insertion point.) Unless you choose to program the idle chain otherwise, unlike the command chain, the whole idle chain is traversed, even after one object's DoIdle method executes successfully.

When MacApp encounters an alien event, it first traverses the command chain, and then calls gHeadCohandler.DoHandleEvent, which attempts to handle the event. Then, just like the command chain, it calls fNextHandler.HandleEvent until one of the methods handles the event or until the chain is finished.

Summary

MacApp continually handles events as the user creates them, assigning instances to do the necessary work. Much of this work is accomplished behind the scenes, in such a way that you need not even know the event occurred. Other events will cause the MacApp code to execute methods that you should have overridden. By now, you should have some idea of what events will cause certain methods to be called, and to which instances those methods will belong.

As a practical matter, you can often tell when you should override a method by its name. The MacApp programmers christened the most important methods to override with names that start with "Do"—DoMouseCommand, DoKeyCommand, DoMenuCommand, and so on.

You should be aware that these event-handling methods are not the only methods you should be concerned about. There are also a great many methods that you will need to write to set up menus, draw the contents of views, create documents and windows, and handle the logic of your application. But this outline should give you a start.

Chapter 6 How to Use MacApp

THE MACAPP PACKAGE CONSISTS OF THE FOLLOWING ITEMS:

- the source code library, including a variety of standard MacApp units and optional building blocks
- several MPW scripts and makefiles to help you automate the compiling and linking of applications created with MacApp
- a number of sample programs

Before using MacApp, you must install it on your hard disk. Most people then build one or more of the sample programs into working applications. Finally, you can start working on your own application, either by starting from scratch or by modifying one of the samples.

MacApp runs under the MPW 3.0 programming environment. To run MPW you need a Macintosh with 128K ROMs and a minimum of 1 MB RAM. If you have not already installed MPW on your hard disk, please refer to the *Macintosh Programmer's Workshop 3.0 Reference*, Chapter 2, "Getting Started," for detailed instructions on how to install MPW. ■

Installing MacApp

This section describes how to install MacApp on your hard disk. It assumes that you have already correctly installed MPW. See the *Macintosh Programmer's Workshop 3.0 Reference* for details on MPW installation. If you are using another development environment, please consult the installation instructions for that environment.

Follow these steps to install MacApp.

1. Create a folder named MacApp in your MPW folder. (If you prefer to place it elsewhere or name it something else, you should modify the MacApp•UserStartup file appropriately to accommodate this. See step 4 below.)
2. Copy the contents of all the MacApp disks into your MacApp folder, and put the disks in a safe place.
3. For every folder with a name in the form More XXX, copy its contents into the folder XXX. Then discard the empty folder More XXX .

   For example, you should take the contents of the folder More MacApp Source Files and put it into the folder MacApp Source Files. Then discard the folder More MacApp Source Files.
4. Place the file User Startup•MacApp in your MPW folder. When you launch MPW, it uses the code listed below to find and execute the the actual MacApp startup file. If you've placed the MacApp folder in a different directory, be sure to modify the first line appropriately.

```
set MacApp "{MPW}MacApp:"
export MacApp
execute "{MacApp}Startup"
```

If you are low on disk space, you can save space with these techniques:

- Delete any fonts, desk accessories, print drivers, utilities, or other elements you don't need.
- Don't copy all the MacApp sample programs onto your disk.
- Change the SeparateObjects flag to FALSE. See "Building a MacApp Program" later in this chapter.

When the installation is done, you should be able to build one of the sample programs or one of your own MacApp programs. Refer to the section "Building a MacApp Program" later in this chapter for details on how to build a program.

File naming conventions

MacApp filenames follow several conventions. Where *AppName* stands for the application name and *UnitName* stands for the name of a particular unit, the constituent files are usually named according to the following rules:

- U*UnitName*.p contains the Pascal interface for the unit U*UnitName*.
- U*UnitName.xxx*.p contains the implementation of the unit U*UnitName*. In most cases, the implementation of a unit is short enough to be put into a single file. In that case, 'xxx' is traditionally replaced with "incl," which stands for "include file #1." However, when the implementation is too long to be managed easily in a single file, it is broken up across a number of files; each is given a mnemonic name. The most notable example of a unit broken into a large number of files is UMacApp, which is broken up into about a dozen files, each named after the most important thing defined in the unit, usually an object class.
- U*UnitName*.a contains supporting assembly source for the unit U*UnitName*, if any is needed.
- M*AppName*.p contains the main program for the application.
- *AppName*.r contains the resource description file (Rez input) for the application.
- *AppName*.MAmake contains the make rules for the application.

If additional units are required, they are usually named in the form U*UnitName*, and the files containing the interface and implementation of the unit follow the conventions given above.

All the MAMake files and USES statements conform to these conventions. The documentation refers to particular files in several ways:

- by filename; for example: UPrinting.p, UMacApp.TApplication.p, and DrawShapes.r
- by the unit name defined within the file; for example: UPrinting, UMacApp, UDrawShapes
- by what the file is; for example: the printing unit, the MacApp unit, and the DrawShapes program

Therefore, the printing unit is defined as UPrinting and can be found in the files UPrinting.p and UPrinting.incl.p. If you use this unit, your USES list would contain the line

```
UPrinting,
```

reflecting both the name of the file and the name of the unit defined in the file.

Please note that while these filename conventions are typical in MacApp, they are not enforced by MacApp. Prefixing a *U* to unit names and an *M* to your main program is a time-honored MacApp tradition and may aid the organization of your files, but it is not mandatory. For instance, calling your program *AppName*.p rather than M*AppName*.p makes no difference to MacApp.

Building a MacApp program

MacApp and MPW provide a powerful and flexible set of tools to help you build MacApp applications. In the simplest case, all you need to do is type `MABuild` and the name of your application, and MABuild will compile and link your code, avoiding recompiling anything that hasn't changed. But when your needs are not so simple, MABuild and the rest of the build system allow you to specify virtually any combination of compile-time options, file interdepencies, names of output files, and a host of other attributes.

We will discuss the basics of building MacApp applications in the following sections; however, you should read the MPW manuals that talk specifically about the linker and about the compilers you choose in order to more fully understand the compile-time options available to you. Also, you should be familiar with MPW's Make command.

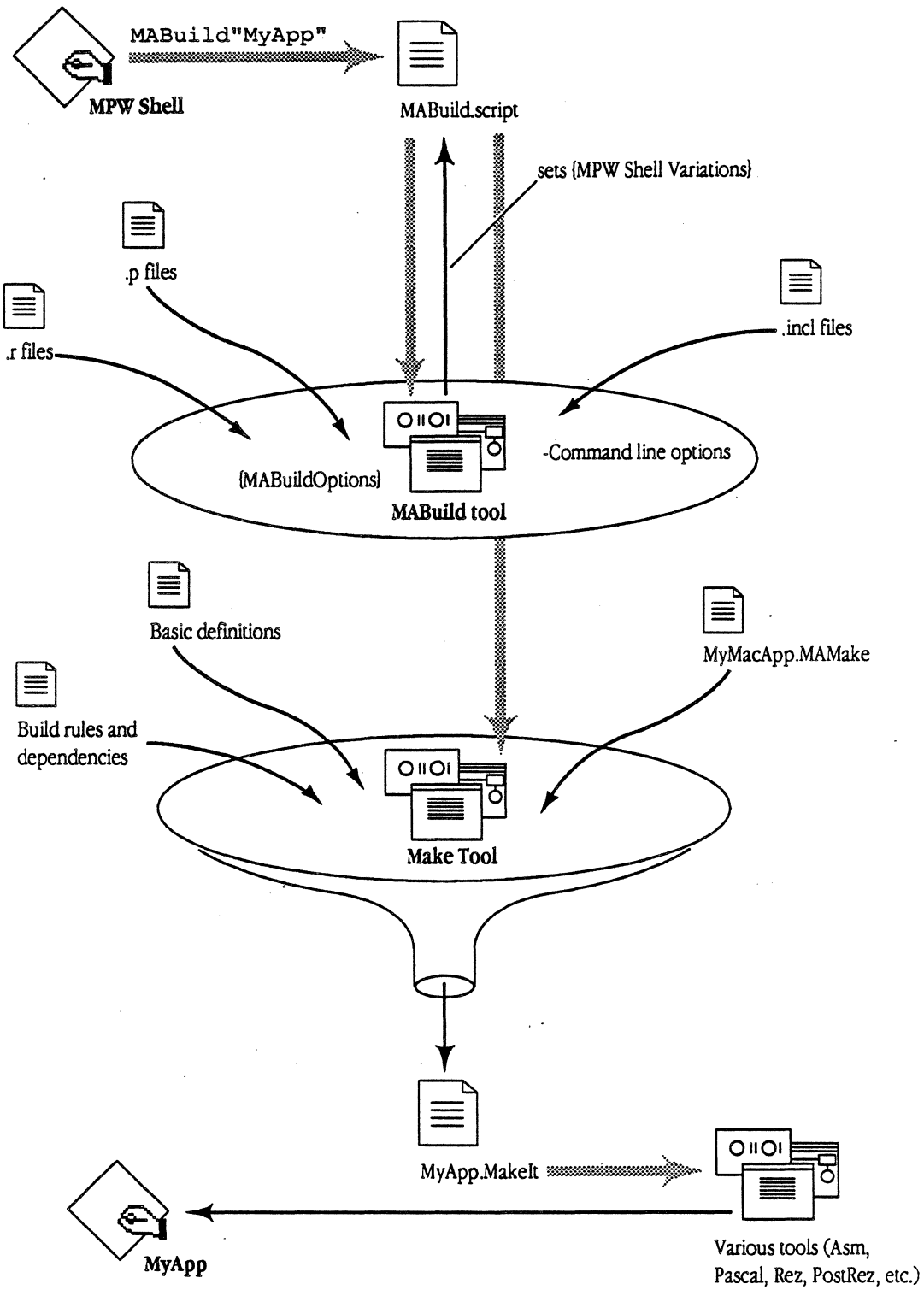
The architecture of MacApp's build system

MacApp's build system involves a complex interaction of MPW scripts, tools, and shell variables. At first, you will need to know little about how those interactions work, since the MABuild system offers quite intelligent default behavior. But once you want to begin exploring the flexibility and power beyond the default behavior, the architecture will become more important.

The system works like this. When you invoke the MABuild command, the MABuild script (called MABuild.script) is invoked. In turn, that script calls the MABuild tool, which sets several local MPW shell variables. The script then uses those local shell variables, some basic makefiles (plus any makefiles you specify), and the MPW Make tool to create the MakeIt file for your application.

Your MakeIt file contains all the commands and options necessary to build your application —and MacApp if necessary. (MABuild and Make are smart enough to avoid recompiling any files that haven't changed since their last compilation.) After creating the MakeIt file, the MABuild script executes that MakeIt file.

■ **Figure 6-1** The MacApp 2.0 Build system



When this process is complete, the MABuild tool, which you invoked with a single command, will have created the MakeIt file (for your reference), the object files (for future builds), and the executable code (so you can test the results). Normally you would not need to be aware of all the behind-the-scenes calling of scripts and tools; however, you should know that MABuild offers some 50 options for your convenience.

In order to avoid forcing you to deal with all of these options every time you need to build an application, the build system offers you defaults in a variety of ways. The MABuild script has some defaults hard-wired into it. It also checks some global MPW Shell variables, which you can set to determine default behavior. When you first install MacApp, a number of those shell variables are already defined in MacApp's startup file. (These are described in the next section.) At first you don't need to pay attention to them, but once you become aware that some other default behavior would be more useful to you, you can alter MacApp's startup file to suit your needs.

Finally, any time you wish to override MABuild's default behavior, you can specify any or all of the MABuild options from the command line. Options set on the command line override all others regardless of whether they come from MABuild's hard-wired behavior or from shell variables set by MPW or by you.

MABuild's input and output

Another way of looking at how MABuild works is to look at its inputs and outputs.

Inputs

There are two sorts of inputs: those you must provide and those already provided when you buy MacApp.

Inputs already on disk. MABuild uses the defaults hardwired into its code and those provided by way of pre-programmed shell variables in MacApp's startup file (unless you modify them). It also uses MacApp's MAMake file to build MacApp's own libraries and two other partial makefiles called Basic Definitions and Build Rules and Dependencies. Of course, it also uses all the source code files and resources that make up the MacApp libraries.

Inputs you must provide. You must provide MABuild with your own source code and (in most cases) your own resource file. In many cases you must also provide a makefile called *YourApp*.MAMake. If you choose, you may also change MacApp's defaults by modifying the shell variables set in MacApp's startup file. And you may override MABuild's default behavior by specifying command-line options.

Outputs

MABuild creates a makefile called *YourApp*.MakeIt which contains a series of MPW commands that will build your application. It also outputs object files for your code as well as MacApp's and creates an executable version of your application.

△ **Important** The *YourApp*.MakeIt script contains the idiosyncratic information for a particular build, excluding files that do not need to be recompiled. Generally, you should not simply re-execute this file the next time you need to recompile. Instead, rerun MABuild so it can check to see what must be recompiled. △

Where does it put all these files? It puts the MakeIt file in the directory specified by the MABuild command, along with the executable object code. These files come in many varieties, with various debugging features on or off, support for different CPUs on or off, and with various optimizations on or off. Unless you specify that you don't care to keep all these variations around, MABuild creates a separate folder for each variation in the target directory, placing each variation of your code's object files and executable application together in a folder named for the options used in the compile.

Similarly, if you are using `-autoBuild` , MABuild creates a separate folder for each variation of MacApp's object files in MacApp's own source code directory, referred to by the shell variable {MALibraries}.

The following section on object folders explains the system for naming these folders. MABuild's options are also explained in more detail later in this chapter. That section also discusses the `-SeparateObjects` flag, which is the option that allows you to specify whether you want all those variations to be kept.

The simplest builds

Generally, MABuild expects you to provide it with a makefile called *YourAppName*.MAMake. This tells MABuild the name of your application, the names of building block interface files you wish to include, and the names of the files that your resource description file includes, as well as a variety of other information. (See "Creating a makefile," later in this chapter.) But there are two cases when you don't need to provide your own makefile: (1) when your application has the classic configuration of a sample program, or (2) when your application is contained in a single file. These conditions are described in more detail in the following sections.

Classic configuration. The classic configuration needed to avoid writing a makefile outlined in table 6-1:

■ **Table 6-1** Configuration for building an application without a makefile

Template for file name	Contents of file
M*YourApp*.p	Main program
U*YourApp*.p	Interface file
U*YourApp*.incl.p	Implementation file (can also be named U*YourApp.xxx*.p with the *xxx* used to number multiple files)
YourApp.r	Resource description file (input to Rez) The resource description file is optional; if you do not provide it, MABuild uses MacApp's default resources)

All of the sample programs except Nothing follow this pattern. When the MABuild script sees that you have not provided an MAMakeIt file, it checks to see if these files exist. If they do exist, it will build the application.

One-file configuration. Another way to avoid having to write your own makefile is to put your main program, interface, and implementation into a single file named *YourApp*.p. (This is recommended for only the smallest applications.) In the absence of a *YourApp*.r file, MABuild will use a default resource file to build the application. This file includes resources to support *all* of the building blocks (such as UDialog or UPrinting) and therefore may be slightly bigger than strictly necessary, but such is the cost of being able to put all your code into a single file.

In these simplest cases, all you need to do is enter the MABuild command with a single parameter, called the target. The target is the name of the application, prefixed by the pathname of the source code directory. Thus, to build the DrawShapes sample, you would enter the following command:

```
MABuild "{MASamples}DrawShapes:DrawShapes"
```

In the example above, the word `MABuild` is the command; the final word, `DrawShapes`, is the name of the application that is to be built, and the rest is the directory pathname. Depending on whether or not you have included debugging code, the application wll be placed in a folder entitled .Debug files or .Non-debug files, respectively, in the DrawShapes folder. Thus, this command builds the application DrawShapes and places it in the proper Debug folder, which is inside a folder named DrawShapes.

◆ *Note:* If {MASamples}DrawShapes:DrawShapes is in the current directory, you need not specify a pathname; however, you may do so if you like.

Other builds

Many builds will not fall into the categories we have dealt with thus far. Perhaps your application has neither the classic file configuration nor the the one-file configuration, or you wish to modify MABuild's default behavior.

If the problem is only that the application does not have a simple file configuration, all you need to do is provide a makefile called YourApp.MAMake, as described in the section "Creating a makefile." (You can see samples of these makefiles in all the sample folders except Nothing.)

If the problem is that you want to override MABuild's default behavior, you have two choices. You can simply specify your choices by way of command-line options. Or, if you know what choices you would make most often, you may want to change MABuild's default behavior by modifying the global shell variables defined in MacApp's startup file. (For a more detailed description of these options and the shell variables that control their default behavior, see "Options for the MABuild script," later in this chapter.)

The syntax for the MABuild command is

```
MABuild [option...] [target...].
```

The options are not case sensitive. The most commonly used build options are described here.

-Debug. By default, MABuild will create a debug version of the code and place it in a folder called .Debug Files. That folder will be placed inside the folder specified as the target of the MABuild command—in other words, it will be inside the folder in which the application's MAMake file resides.

-NoDebug. To create a shippable application, you will want to turn off MacApp's debugging features. This will make the executable application smaller and will make it run faster. To turn off debugging, specify the `-NoDebug` flag. Your object files and executable application will appear in a folder called .Non-Debug Files.

-Sym. To create a version of your application that is compatible with Apple Computer's symbolic debugger, SADE™, you should specify the `-Sym` flag. Your object files and executable application will be in a subdirectory called .SADE Debug Files. If you want to create a version compatible with SADE but without MacApp's own debugging versions turned on, you should specify both the `-Sym` option and the `-NoDebug` option. Object files and applications created with both those options will be placed in a subdirectory called SADE Non-Debug Files. The `-Sym` option is turned off by default, unless you change MPW's User Startup file to specify otherwise.

Creating a makefile

While simple MacApp programs may be built using MABuild's default configuration, most applications created with MABuild will need to have a makefile with a name in the form *AppName*.MAMake, where *AppName* is the name of the application. In this file you specify the application name, the signature, and what files your application depends on.

The file Application.MAMake is included in the MacApp 2.0 release and is a template for your program's makefile. Most of the file is simply a set of assignments to Make variables. The variables are summarized in Table 6-2:

■ **Table 6-2** Make variables

MPW Shell variable	Definition
`AppName`	the name of the application
`NeededSysLibs`	a list of required system libraries
`BuildingBlockIntf`	a list of the building block interface files the application uses
`BuildBlockObjs`	a list of the building block object files the application links with
`OtherInterfaces`	a list of other Pascal interfaces the application uses
`OtherLinkFiles`	a list of other object files the application links with
`OtherSegMappings`	a list of -sn (segment name) commands for the linker
`OtherRezFiles`	a list of .r files that must be included in the Rez command, other than *AppName*.r
`OtherRsrcFiles`	a list of resource files that are included by one of your Rez files, other than the Debug.rsrc and MacApp.rsrc files

In the simplest case, all you will have to do is perform a global search-and-replace on a valid makefile, replacing the old application's name with the name of the new one. (See the makefiles for the sample programs if you need additional guidance.)

If you want to specify any additional files (such as assembly language files) that need to be linked with your application, you must provide dependency rules for them in your makefile. For example,

```
OtherLinkFiles = "{OBJAPP}"MyApp.a.o

"{OBJAPP}"MyApp.a.o      ƒ "{SRCAPP}"MyApp.a "{SRCAPP}"FooInclude.a
```

You do not need to specify build rules for Pascal compiles or MPW assemblies because they are covered by the default rules specified in Build Rules and Dependencies. (Note that the *ƒ* character is produced by typing Option-f on the keyboard.)

The MABuild command file calls Make with your specified makefile and the MacApp makefiles. Make uses files in theo following order: Basic Definitions, *YourApp*.MAMake (if supplied), and Build Rules and Dependencies.

Options for MABuild

Depending on how you count them, MacApp has between 50 and 100 command-line options. They range from ones that get passed directly to the compilers to those that control the level of diagnostics printed while script is running. Each option has a default hard-wired into the script, and that default can, in turn, be modified by setting a global MPW shell variable. Sometimes these two sorts of defaults are in opposition. For instance, the MABuild script's default determines that diagnostic information will not be printed. But when you first install MacApp, the startup script sets a global variable overriding that default. In short, any command-line options you explicitly specify override options specified by shell variables; shell variables, in turn, override any defaults hard-wired into the code.

Specifying options

When specifying options, keep the following in mind:

- All options may be specified on the command line.
- You may specify as many options as you like. To continue a command on the next line, type the continuation character "∂" (Option-d) and press Return.
- Each option is preceded by a dash, and case is not significant. Some options take parameters, however, and case in parameters may be significant.
- If the parameter has a space in it, you must enclose all the parameters in single or double quotation marks. If you aren't familiar with the use of quotation marks as they relate to the command-line parser in MPW, read the section "Quoting Special Characters" in the *Macintosh Programmers Workshop 3.0 Reference.*

Some options listed in Table 6-3 begin with `[No]`. Each option controls a flag that can be turned on (if specified without the "No") or off (if specified with the "No"). For example, if you specified the `-Run` option, MABuild would execute the application after linking it; if you specified the `-NoRun` option, it would not.

■ **Table 6-3** MABuild options

Option	Default in Script	Default in Original Startup File	Description
-Asm *option...*	(Empty)	'-case on'	Options to pass through to the assembler
-C *option...*	(Empty)	(Empty)	Options to pass through to the C compiler
-d *name*=(TRUE|FALSE)	(Empty)	(Empty)	Set compile-time variable *name* to TRUE or FALSE
-Lib *option...*	(Empty)	(Empty)	Options to pass through to the librarian
-Link *option...*	(Empty)	(Empty)	Options to pass through to the linker
-Make *option...*	(Empty)	-w	Options to pass through to the make tool
-p	On	On	Simple progress indication to standard output
-Pascal *option...*	(Empty)	(Empty)	Options to pass through to the Pascal compiler
-Rez *option...*	(Empty)	-rd	Options to pass through to the Resource compiler
-PostRez *option...*	(Empty)	(Empty)	Options to pass through to the PostRez tool
-E	(Empty)	(Empty)	Pass the -E option to the Make tool, causing it to rebuild everything that is part of the target
-S	(Empty)	(Empty)	Pass the -S option to the Make tool, causing it to write to standard output a dependency graph for the target See also the -R option, below.
-R	(Empty)	(Empty)	Pass the -R option to the Make tool, causing it to write only the roots of the dependency graph to standard output. See also the -S option.
-RenameFlag *oldname newname*	(Empty)	*	Rename the combination of flags *oldname* to *newname*
-[No]Align	On	(Empty)	Longword align all code and data for speed
-[No]AutoBuild	Off	Off	Auto-build the MacApp libraries (and any others specified in the environment variable {MAAutoBuild})
-[No]BottleNeck	DND	DND	Bottleneck method calls through the dispatcher
-[No]Debug	Off	On	Turn on debugging
-[No]Execute	On	On	Execute the results of Make
-[No]ExpandEnvVars	Off	Off	Expand environment vars in *AppName*.makeit
-[No]Fail	On	(Empty)	Stop executing after first failure
-[No]Inspector	DND	DND	Include inspector in debugging gear

(continued)

■ **Table 6-3** MABuild options (continued)

Option	Default in Script	Default in Original Startup File	Description
-[No]MacApp	On	(Empty)	Build for use with MacApp
-[No]Names	DND	DND	Include embedded debugging symbols
-[No]NeedsColorQD	Off	Off	Build to require Color QuickDraw
-[No]NeedsMC68020	Off	Off	Build to require a 68020 or better
-[No]NeedsMC68030	Off	Off	Build to require a 68030 or better
-[No]NeedsFPU	Off	Off	Build to require a floating point unit
-[No]NeedsROM128K	Off	Off	Build to require 128K ROM or better
-[No]NeedsSystem6	Off	Off	Build to require System 6.0 or better
-[No]Perform	DND	Off	Include performance monitor code
-[No]PP	Off	Off	Have all tools show progress
-[No]ProceduralViews		Off	Off Include code to create views procedurally
-[No]RangeCheck	DND	Off	Force rangechecking
-[No]Run	Off	Off	Run after build
-[No]Save	Off	Off	Auto-save all open windows before starting build
-[No]SeparateObjects	On	On	Keep separate object folders for each build variation
-[No]StatusOnly	Off	Off	Write results of Make to StdOut
-[No]Sym	Off	Off	Generate SADE-compatible symbol information
-[No]T	Off	Off	Have all tools show elapsed time
-[No]TemplateViews	On	Off	Include code to create views from templates
-[No]Trace	DND	DND	Include tracing facilities in debugging gear
-[No]UnInit	DND	DND	Force initialization of uninitialized storage
-[No]UserAutoBuild	Off	Off	Auto-build user-specified libraries {MAUserAutoBuild}
-[No]WriteLnWin	Off	Off	Include writeLn window in debugging gear
-[No]WriteTemplates	Off	Off	Include code to create write views to templates

* The `-RenameFlag` option allows you to specify different names for the folders in which your object files and executable applications go. MacApp's startup file defines four of these: Debug files, SADE Debug files, Non-Debug files, and SADE Non-Debug files. See the section below that discusses object folders.

DND Means an option is On in Debug mode, Off in NoDebug mode.

Object folders

MacApp allows you a vast array of compile-time variables that you can set using MABuild's flag options. By setting the appropriate option, you may keep a separate folder for each set of object files with a different combination of compile-time options. The folders will be subdirectories of the folder in which you keep the source code.

MABuild keeps separate versions of the folders only if its `-SeparateObjects` flag is set to TRUE, the tool's default state. If you do not want to keep all of these versions, perhaps because of limitations on disk space, you should either specify `-NoSeparateObjects` on the command line or edit MacApp's startup file to take out the `-SeparateObjects` line.

How does MABuild name the folders in which it keeps these separate versions? It uses a strict convention that assigns a two-character mnemonic for each compile-time flag that is TRUE for a particular version, according to the table below. Then it strings together all the pairs for one long name.

Although these names indicate exactly what's in the folders, they are not easily parsed by human eyes, so MABuild offers an option (the `-RenameFlag` option) allowing you to give any name you like to any particular set of flags. In fact, MacApp's startup file as it is provided by Apple defines remappings for four of the most popular sets of flags:

```
-RenameFlag NmDbInUnPeRaWrTrPrTe 'Debug Files'∂
-RenameFlag NmDbInUnPeRaWrTrPrTeSm 'SADE Debug Files'∂
-RenameFlag PrTe 'Non-Debug Files'∂
-RenameFlag PrTeSm 'SADE Non-Debug Files'"
```

If you prefer the more conventional names, you can erase these lines from MacApp's startup file. To define more names, you can either specify the `-RenameFlag` option on the command line or include the option with the other four in MacApp's startup file. For example, if you wanted to build DrawShapes with procedural views support (the `-Pr` option), template views support (the `-Te` option), and WRITE template views support (the `-Wt` option), you would invoke MABuild followed by those options and a target, like this:

```
MABuild PrTeWt "{MASamples}DrawShapes:DrawShapes"
```

Here we are telling MPW to look for source code in the directory specified by the shell variable {MASamples}, build DrawShapes with the options we specified on the command line, and place the output in a folder called DrawShapes, which itself is located in the folder DrawShapesFld. Remember that MABuild will also invoke any other build options specified in the build script or other shell variables. Should any of these options be in conflict with those specified on the command line, the options specified on the command line will take precedence.

If you find that you want to use this combination of options frequently you can use the `-RenameFlag` option to name your set of options something with easier to remember, such as MyBuildOpts, like this:

```
-RenameFlag PrTeWt 'MyBuildOpts'
```

By placing this line in MacApp's startup file you can then specify this set of compile-time options by simply calling MyBuildOpts on the command line, as follows:

```
MABuild {MyBuildOpts} "{MASamples}DrawShapes:DrawShapes"
```

You can further simplify the process by modifying the appropriate lines in MacApp's startup file to always invoke these particular options, thus making your preferences be MABuild's default behavior. Then you need only invoke the MABuild command with a target, as in our original DrawShapes example. MABuild's defaults will already have been set to your specifications at startup.

Each of these three ways of setting compiler options will produce the same result; the matter is simply one of preference. Table 6-4 is a complete list of compiler mnemonics.

■ **Table 6-4** Two-letter compiler option mnemonics

Code	Mnemonic for	Compiler option
`Db`	Basic debugging gear	qNames qDebug qInspector qUnInit qPerform qRangeCheck qWriteLnWin qTrace
`64`	Do *not* require 128K ROM	qNeedsROM128K
`S6`	No support for Systems earlier than System 6.0	qNeedsScriptManager qNeedsHierarchicalMenus qNeedsStyleTextEdit qNeedsWaitNextEvent
`Cq`	Require Color QuickDraw	qNeedsColorQD
`20`	Require 68020 CPU	qNeedsMC68020
`30`	Require 68030 CPU	qNeedsMC68030
`Fp`	Require 68881 coprocessor	qNeedsFPU
`Nm`	Embed debugger names	qNames
`Db`	Debugging support	qDebug
`In`	Inspector support	qInspector
`Un`	Uninitialized storage support	qUnInit
`Pe`	Perform support	qPerform
`Ra`	Do range checking	qRangeCheck
`Wr`	WriteLnWin support	qWriteLnWin
`Tr`	Trace support	qTrace
`Pr`	Procedural views support	qProceduralViews
`Te`	Template views support	qTemplateViews
`Wt`	WRITE template views support	qWriteTemplates
`Sm`	SADE (MPW symbolic debugging) support	qSym

MPW shell variables for MABuild

The MABuild script looks at a number of global shell variables to determine its default behavior. The shell variables are set in MacApp's Startup file. Since most of them describe the name or location of source files and tools, you should not change them without careful consideration. A few, however, are there especially for your convenience:

{MABuildDefaults} These are all the options you may want to call every time you invoke the MABuild command. For example, if there were an option you would specify 60 percent of the time, you could include it here. Then, for those times you didn't want it, you could specify the opposite on the command line. Options specified on the command line always override those supplied by shell variables. See the startup file to see how these variables are originally configured.

{MAUserAutoBuild} This variable contains a list of targets you always want appended to any targets you include on the command line. This is useful if you have added your own library of objects to MacApp's and want to make sure they are kept up to date. However, the drawback is that every time you build you could lose 20–30 seconds of checking to see whether these files need to be recompiled. If your library is not likely to change, you may be better off not using this option. This variable is empty as supplied in the MacApp 2.0 release.

{MADone} This variable specifies what MPW command should be executed when the build is successful. The default for this command is to play a series of beeps. You may modify it to run silently, as with all MPW commands.

{MAFailed} This variable specifies what MPW command should be executed when the build fails. The default for this command is to play a series of beeps. You may modify it to run silently, as with all MPW commands.

MacApp Rez files

While it is possible to build a simple MacApp application without a resource description file, using only the defaults specified in the MPW Shell variables, the MABuild script and the default .r file, it is more likely that you will need to have one or more of these resources in a *YourApp*.r input file to Rez (the MPW resource compiler):

- MacApp Include files, which are described later in this chapter
- an Include line for your code (see the sample programs' resource files for examples)
- a `'cmnu'` resource for your menus, which is described later in this chapter
- an `'MBAR'` resource, described later in this chapter
- an "About application" dialog (see the sample programs' resource files for examples)

There are three other resources that are not required, but strongly recommended, for compilation:

- a `'SIZE'` resource to tell MultiFinder how much space your application will need
- the `'seg!'` and `'mem!'` resources to provide information for efficient memory management

For complete examples of any of these elements, look at the *AppName*.r files of the sample programs.

Include files

Most of the standard MacApp resources are supplied as *AppName*.r (Rez input) files that are compiled by MABuild. You include the compiled versions with lines like

```
include "FileName";
```

where *FileName* is the name of a MacApp file. If you don't need all the standard resources, you can copy the files and remove the resources you don't want.

Some resources required by MacApp must be in your resource description (.r) file. These include any `'view'` resources you may have created as well as any `'cmnu'` or `'MENU'` resources your application will require. (You would use `'MENU'` resources only in rare cases, if you want to bypass MacApp's menu handling.) Look at the sample programs for guidance.

The 'cmnu' resource type

The 'cmnu' resource type is provided so that menu items can be identified by a unique command number, independent of the item's placement within the menus.

You define resources as type 'cmnu' instead of type 'MENU'. These have the same syntax as resources of type 'MENU', except that menu items have one additional component, a command number (or the symbol `nocommand = 0`). After running Rez, the makefile runs the tool PostRez, generating the usual 'MENU' resources as well as a single resource of type 'mntb' and ID 128. This particular resource maps all command numbers for all 'cmnu' resources encountered in the file into their corresponding menu and item numbers.

△ **Important** Because the PostRez tool changes 'cmnu' resources to 'MENU' resources and other information MacApp needs for handling menus properly, you cannot currently use DeRez to decompile cmnu resources for MacApp programs. In addition, you should not use the resource editor to change MENU resources, because that will not provide all the information MacApp needs. To change your menus, you must edit the original resource description file (*MyApp*.r file) and allow MABuild to run Rez and PostRez again. △

'MBAR' resources

MacApp requires one or two 'MBAR' resources that tell it what to do with your menus.

You must have an 'MBAR' resource with ID 128. This resource gives the ID's of all menus that MacApp should read in and display initially. MacApp looks for this resource when it starts up and loads all menus whose ID's are listed here.

You can also have an 'MBAR' resource with ID 129. This resource gives the ID's of menus that can be added to or removed from the menu bar by your application. The menus listed here are read in by MacApp but are not initially displayed.

You can also have menus that are not listed in either 'MBAR' resource.
Examples of these are

- buzzword menus, which contain strings meant to be placed in other menus under certain conditions
- the Debug menu, which has ID 900 and is treated as a special case by MacApp so that it is only read in if debugging code is included in the application

- nonstandard menus such as graphical menus. These must have ID's greater than mLastMenu (which is 63). You need to handle these menus by calling the Menu Manager directly

Menus handled by MacApp must have IDs from 1 to mLastMenu, regardless of which 'MBAR' resource refers to them.

△ **Important** You should not use programmatically created menus; that is, your resource file should contain at least the menu title of every menu. △

The structure of a MacApp program

MacApp programs are written using a certain structure. Some elements are mandated by the syntax and are identified here as such, and others are simply conventions that make it easier to maintain the programs.

The elements of a MacApp program are the main program, the unit's public interface, and the unit's implementation. Each element is generally kept in a separate file, although you can combine them if you wish. See the sample programs for more detailed information on these files.

The main program is generally very simple and small. According to convention, it is usually named M*AppName*.p where *AppName* is the name of the application, although you can name it anything you wish as long as the name ends in *.p*. A MacApp main program follows this structure:

```
PROGRAM YourName;
    USES                                        {Units used by this program}
      UMacApp,            {Typically you would include the UMacApp unit}
      UPrinting,    {You might also include one or more Building Blocks}
      UDialog,                                   {like UPrinting or UDialog}
      Fonts;                                          {Implementation Use }
    VAR
      gYourApplication: TYourApplication;              {The application object}
    InitToolBox                    {Initialize the Toolbox and utilities};
    IF ValidateConfiguration(gConfiguration) THEN
                         {MacApp verifies that the host system has the }
                                    {requirements to run your program }
      BEGIN                  { Continue with remainder of initialization }
      InitUMacApp({Number of master pointer blocks to allocate});
      InitUPrinting;                       {Initialize the UPrinting unit }
                                             { if you are using UPrinting }
      InitUTEView;                         {Initialize the UTEView unit }
                                             { if you are using UTEView }
      InitUDialog;                         {Initialize the UDialog unit }
                                             { if you are using UDialog }
      InitUGridView;                       {Initialize the UGridView unit }
                                             { if you are using UGridView }
      New(gYourApplication);          {Allocate a new TYourApplication object}
      FailNil(gYourApplication);     {Signal if New fails from lack of mem}
      gYourApplication.IYourApplication(kFileType);     {Initialize the new  }
                                                         {application object }
      gYourApplication.Run;  {Run the application; hand control to MacApp}
      END
    ELSE
      StdAlert(phUnsupportedConfiguration);  {Alert if minimum system }
                                          { requirements are not present}
END.
```

The rest of the application is generally run from the application's Run method. The basic type TApplication is defined for you by MacApp, and you customize it in one of the units to create your application type, called T*YourApplication* here. That unit is listed in the USES declaration along with any other units needed to compile its interface.

The unit's public interface file, usually called U*AppName*.p, defines the object types used by the program. The methods of object classes must be declared in the style of a FORWARD declaration (although you never use the reserved word FORWARD). Where you want to reimplement an inherited method, you write the full interface for the method and follow it with the OVERRIDE reserved word. A unit interface file usually follows this structure:

```
UNIT UnitName;
INTERFACE
    USES                                        {Units used by this program}
      UMacApp,            {Typically you would include the UMacApp unit}
      UPrinting,    {You might also include one or more Building Blocks}
      UDialog,                                   {like UPrinting or UDialog}
      Fonts;                                           {Implementation Use }
    CONST
                                      {Any constants for the application}
    TYPE
                                             {Ordinary type declarations}
                                               {Object-type declarations}
IMPLEMENTATION
    {$I UName.inc.p}                  {Name(s) of implementation file(s)}
END.
```

The unit implementation file, usually named UAppName.inc1.p, can begin with a VAR part that declares any global variables for the unit followed by the implementations of the methods. You have the option of repeating the full interface declaration with the implementation of each method. There is nothing special about the design of this file; it consists simply of the global variable declarations, private constructs and type declarations, and the method implementations, which look like procedure and function declarations. If an application is large, its implementation can be broken up into a number of units, naming them U*AppName.xxx*.p.

MacApp's debugging features require the use of some compiler switches that you can set from the MPW or MacApp startup file, from MPW Shell variables, or directly from the command line.

Chapter 7 MacApp Debugging Facilities

MACAPP PROVIDES SEVERAL TOOLS TO HELP YOU DEBUG YOUR APPLICATION and fine-tune its performance. These include the Debug window, the Debug menu, Inspector windows, and MacApp's Interactive Debugger.

- **The Debug window.** This window is available when the MacApp code used with the application was compiled with debugging code included and allows you to use the Debug menu and the Interactive Debugger. Output from Write or WriteLn statements are directed to this window. Input may come from Read and ReadLn statements or from the command line in debug mode.
- **The Debug menu.** This menu appears in the menu bar of any MacApp application (compiled with debugging code included) that has the Debug menu in its resource file. Using this menu, you can control certain features of your application and cause certain information to be printed instantly or periodically in the Debug window.
- **Inspector windows.** These windows provide an easy way to display the fields of any object instance. An Inspector window contains a list of every object class that has at least one instance. When you choose one of these classes, a list of every instance of that class appears. You can then choose to see the fields of any of these instances. Since you can have multiple Inspector windows, you can examine the fields of more than one instance at a time.
- **The Interactive Debugger.** This is a high-level debugger that runs in the Debug window. It takes control whenever there is an error, as well as at the beginning and end of most MacApp methods and routines. Unless you specify otherwise, it also takes control at the beginning and end of methods and routines in your application.

Also available to you are Macsbug (the low-level Macintosh operating system debugger) and all of MPW's performance-reporting tools. While we will discuss the use of Macsbug and MPW performance-tuning tools with MacApp, it is recommended that you consult the *Macsbug 2.0 Reference* and the *Macintosh Programmer's Workshop 3.0 Reference* for further information. ■

MacApp's debugging gear

To use the MacApp debugging facilities, your application and the version of MacApp that you use must have been compiled with debugging on. See Chapter 6, "How to Use MacApp," for information on how to turn off debugging. (The easiest way to tell whether or not a version has been compiled with debugging on is by its size: Debugging code typically doubles the size of the MacApp object file. Note, though, that additional building block units such as UTEView, UPrinting, and UDialog also make the object file significantly larger, so you can only compare files with the same units.)

None of the debugging facilities are available when MacApp is compiled without debugging code; however, you can always use the low-level MPW debugger, MacsBug, with your MacApp application—even if it's compiled with debugging off. Some information about using MacsBug with a MacApp application is included under "Using MacsBug with MacApp," later in this chapter. See the *MacsBug 2.0 Reference* for complete information on MacsBug.

You should always recompile MacApp and your application without debugging code when you want to produce a production version of your application. For information, see "What controls debugging code" in this chapter.

- *Note:* To have the full set of debugging facilities, you must build your application using the `-Debug` flag. This will link your application with MacApp's debugging gear. These units are normally linked if you use the default build files. To have the Debug menu appear, you must have that menu in the resource file. It is included in the MacApp debugging resource file. Applications don't need to have USES UDebug, UTranscriptView, or UInspector themselves unless they call the routines in these units directly. (Note that you can call WriteLn without USES UTranscriptView.)

What controls debugging code

Several factors control what you can do with the debugging facilities. They are described in this section.

Compiler variables

There are a number of conditional compilation variables defined in MacApp that are always TRUE when the code is compiled with debugging on and always FALSE when the code is compiled with debugging off. (You can also set their values individually from the compiler command line, although most users find this less convenient.) The table below lists compiler flags that set debugging gear options.

■ **Table 7-1** Compile-time debug flags

Flag	Function
qNames	Generate embedded names (forced TRUE in Debug mode)
qDebug	Turn debugging on or off
qInspector	Turn Inspector window on or off
qUnInit	Use Pascal support to initialize storage (stack, local vars, globals) to a known value
qPerform	Include performance-measurement tools (makes them available but does *not* turn them on)
qRangeCheck	Use Pascal's range-checking code for parameters, strings, variables, and array boundaries. Also use Pascal's overflow checking.
qTrace	Enable or disable tracing in debug mode

To insert code that will be compiled only when one of these switches is TRUE, precede the code with a line such as

```
{$IFC variable}
```

where *variable* is replaced by the identifier of one of the compiler variables. Follow the code with

```
{$ENDC}
```

Everything between these switches will be compiled only when the variable is TRUE. In general, you will use the qDebug switch to delimit your debugging code.

The $D switches

The $D compiler switches {$D++} and {$D+} cause the compiler to insert additional debugging information in the compiled code.

When you use {$D+} or {$D++}, the compiler inserts a variable-length identifier after the return instruction of every method or other procedure. (Sixteen characters are used for a method; eight are used for an ordinary routine).

In addition, when you use {$D++}, the compiler generates a call to the Interactive Debugger at the entry and return of every routine and at every EXIT statement. When the Debugger is called in this way, you can stop the program with the attention keys, or the Interactive Debugger can stop the program because of a breakpoint or because you had run the program with the Single Step command (see "Entering debug mode," later in this chapter).

When you use {$D+}, the compiler inserts the identifier but does not generate any calls to the Interactive Debugger.

In either case, the identifier allows the Interactive Debugger to display the routine names for debugging.

Each switch affects every routine until the other switch or {$D–} is encountered.

When you are debugging, you generally want every routine in your code to be covered by {$D++}. UMacAppUtilities includes the following switch:

```
{$IFC qTrace}{$D++}{$ENDC}
```

Because of this switch, every routine in a unit that names UMacAppUtilities in its USES clause (including all of MacApp) is automatically covered by {D++} unless you use {$D–} or {$D+}. (See the *MPW 3.0 Pascal Reference* for an explanation of these switches.)

If you want a particular routine to be excluded, embed it as follows:

```
{$PUSH}{$IFC qTRACE}{$D+}{$ENDC}
PROCEDURE Example;
.
.
.
BEGIN
.
.
.
END;
{$POP}
```

See also `$PUSH` and `$POP` in the *MPW Pascal 3.0 Reference.*

You can similarly embed a group of routines in the same switches.

Note that the use of {$D-} alone is not recommended. Even with thoroughly debugged routines, it is better to have the routine's name available. (In production code, the names are not included because the qNames compiler variable is set to FALSE.)

△ **Important** Assembled routines (such as those described in *Inside Macintosh*) or routines from units that do not use UMacApp (such as PasLib) are never governed by the $D switches. △

Including debugging code

An application often includes code that is used only for debugging. MacApp uses the compiler variable qDebug to control compilation of debugging code. To insert code that will be compiled only when debugging is desired, surround the code with

```
{$IFC qDebug}
```

and

```
{$ENDC}
```

All code within these markers is compiled only when qDebug is TRUE.

◆ *Note:* You can also write code so that, when debugging, you can enable or disable features. This ability is described under "The Experimenting flag" portion of "Interactive Debugger commands" later in this chapter.

The Debug window

The Debug window is available when you start up any application that is linked with a version of MacApp compiled with debugging on. It is a scrollable, movable, resizable window. It is initially inactive. If you intend to use it often, you may want to position it and your other windows so the Debug window is always visible.

Application mode and debug mode

When debugging code is included in a MacApp program, the program has two distinct modes: **application mode** and **debug mode.** In application mode, the application runs normally (except for any special behavior you insert with debugging code), and the Debug window can be activated, resized, moved, and scrolled like any window. The application may use WriteLn to write information in the Debug window, and information may be written there in response to Debug menu commands or Interactive Debugger commands.

In Debug mode, most user commands are not processed; your interface to the application is through the Debug window's command line. You do have limited use of the mouse and menus, mainly for seeing which menu items are enabled, and application switching under Multifinder is still available through the Apple menu. For more information, see the section "Entering debug mode," later in this chapter.

Using WriteLn statements with the Debug window

There is no special trick to using WriteLn statements and Write statements to print in the Debug window. MacApp uses UTranscriptView, a unit provided with MacApp, to set up the debugging window so that all WriteLn text goes to that window. (Note that, except in debugging, you should never use WriteLn to print text in a MacApp program. Instead, use UTEView or the QuickDraw text calls.)

You always enclose WriteLn statements with conditional compilation switches so that they are not included in production code. See "Including debugging code," above.

Reading debugging information in application mode

If you want to type debugging information, you can use Read and ReadLn. As with WriteLn, you normally surround Read and ReadLn statements with conditional compilation switches so they will not be included in production code. See "What controls debugging code" earlier in this chapter.

The Debug menu

Any MacApp application that was compiled with debugging on and includes the MacApp debugging resources has the Debug menu displayed in its menu bar. The Debug menu lets you control certain features of your application and print certain information in the Debug window. This section describes the commands in that menu.

The rest of this section details the standard Debug menu commands. You can add additional commands by editing Debug.r and adding support for your commands. You may want to add your own commands in an additional Debug menu, rather than adding them to the MacApp Debug menu. (See the sample programs for examples.)

New Inspector Window

This command creates a new Inspector window titled "Inspector Window *N*." You will want one Inspector window whenever you want to examine the fields of object instances. You might want multiple Inspector windows to examine fields of more than one instance simultaneously.

Make Front Window Modal

This is a toggle command. It controls the setting of the front window's fIsModal flag. If the front window is already modal, this command reads "Make Front Window Modeless." You would use Make Front Window Modeless if you had a modal dialog up and wanted to activate an Inspector window. Once you had done what you wished in the Inspector, you could use Make Front Window Modal to return to the previous state.

Do First Click For This Window

This is a toggle command. Choosing it toggles the front window's fDoFirstClick field. If fDoFirstClick is TRUE, this reads "Don't Do First Click For This Window" and the mouse click that activates the window is also passed to gTarget.DoMouseCommand for action, assuming the click is in the content area of the window. When fDoFirstClick is FALSE, the first mouse click activates the window but is not passed to DoMouseCommand.

Refresh Front Window

Choosing this command invalidates the entire front window, resulting in an update of the contents (as well as the borders) of the window. As an alternative to selecting this menu command you can type command-option-R.

Allow Trace Of Menu Setups

This is a toggle command. When it is checked and the Interactive Debugger is in trace mode (see "The Trace command (T)," later in this chapter), {$D++} program points are printed in the Debug window even during the menu setup cycle.

With this option enabled,the Debug window will show the enabled or disabled state of menu items as they are changed by your DoSetupMenus methods, in addition to its normal display.

Allow Trace During Idle

This is a toggle command. When it is checked and the Interactive Debugger is in trace mode (see "The Trace command (T)," later in this chapter), {$D++} program points are printed in the Debug window even during the idle cycle.

Enter MacApp Debugger

This menu command or its command key equivalent, command -`, drops you into the debugger at the next {$D++} program point. For more information , see the section "Entering debug mode" later in this chapter.

Show Debug Window

When you choose this command and the Debug window is closed, the Debug window is displayed. If the Debug window is covered by another, this command brings it to the front and updates the debug information. When the Debug window is open, this command toggles to "Hide Debug Window."

Show Software Version

Choosing this command results in a call to the method gTarget.IdentifySoftware. That method normally prints the compilation data and time in the Debug window.

Inspector windows

Inspector windows provide an easy way to display fields of your program's instances, as well as a few other types of information. Inspector windows are created by choosing the New Inspector Window command from the Debug menu. You can create multiple Inspector windows, which allows you to examine the fields of more than one instance at once.

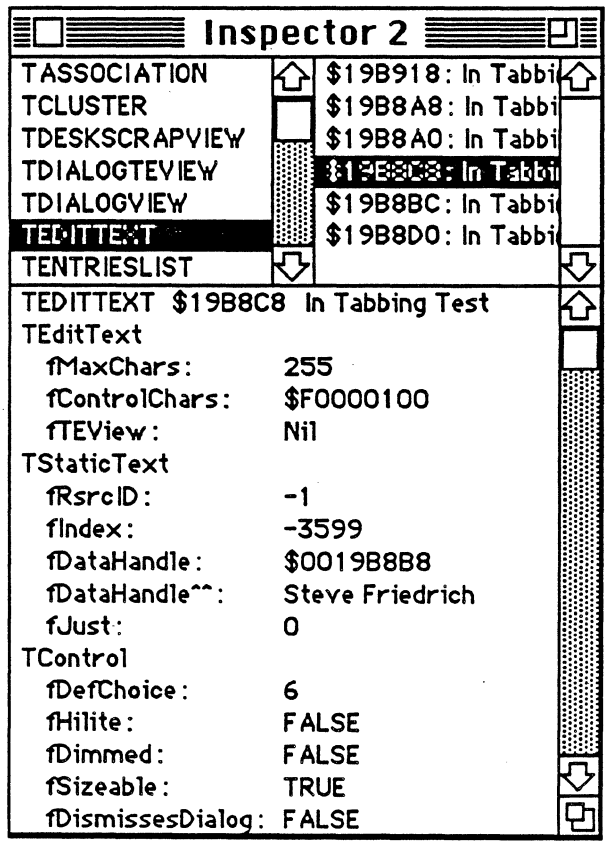
Using Inspector windows

The upper-left pane of an Inspector window is a list of classes that have at least one instance. Clicking one of the class names fills the upper-right pane with a list of instances of that class. Clicking one of the objects causes the lower pane to display the fields of that object. Once an object is displayed in the lower pane, you can also click a field that is one of the following types: an object reference field, a GrafPtr, a WindowPtr, a ControlHandle, a TEHandle, or a RgnHandle. After you click one of these fields, it will be displayed in the lower pane. Incidentally, holding the option key while clicking a field causes that field to be displayed in a new Inspector window, leaving the original unchanged.

The lower pane is *not* automatically updated when the information being displayed changes; however, when the Inspector window is redrawn, the new information is displayed. You can redraw the contents of all of the currently open Inspector windows by choosing the Refresh Inspector Windows command from the Debug menu. Alternatively, you could redraw a single Inspector window by clicking on it and choosing the Redraw Front Window command from the Debug menu.

Figure 7-1 shows a sample session with an Inspector window.

■ **Figure 7-1** A sample session with an Inspector window



There are two methods you will need to override for *each* of your object classes if you want to fully employ the Inspector window: the Fields method and the GetInspectorName method. Both of these methods have been added to the TObject class.

The Fields method

The Fields method returns information about each field of an object. It has already been implemented for all of the classes defined by MacApp, so you will only need to implement it for the classes you define.

- *Note to MacApp 1.1 programmers:* You should replace with Fields methods any Inspect methods you may have defined.

The declaration line of Fields looks like this:

```
PROCEDURE TObject.Fields (PROCEDURE DoToField(fieldName:  Str255;
                                              fieldAddr:  Ptr;
                                              fieldType:  integer));
```

In general, a Fields method should call the DoToField procedure on each field defined by its class. Typically, a Fields method will first call DoToField to report the class name, then call DoToField for each field in the class, and finally call the inherited Fields method, which will report the inherited data.

For example, imagine that you've defined a TShape class like this:

```
TShape = OBJECT(TObject)
      fRect:   rect;
      fColor:  RGBColor;

      {$IFC qDebug}
      TShape.Fields(PROCEDURE DoToField(fieldName:  Str255;
                                        fieldAddr:  Ptr;
                                        fieldType:  integer); OVERRIDE;
      {$ENDC}
END;
```

You should implement its Fields method like this:

```
PROCEDURE TShape.Fields (PROCEDURE DoToField(fieldName:  Str255;
                                             fieldAddr:  Ptr;
                                             fieldType:  integer);

BEGIN
      DoToField('TShape', NIL, bClass); { First report the class name  }
      DoToField('fRect', @fRect, bRect);{ Then report the fields  }
      DoToField('fColor', @fColor, bRGBColor);
      INHERITED Fields(DoToField);      { Finally report the inherited }
                                        { fields.}
END;
```

As you can see, DoToField has three parameters: fieldName, fieldAddr, and fieldType.

- The fieldName parameter is a string containing the name of the data (or class) that you want reported.
- The fieldAddr parameter is a pointer containing the address of the data that you want reported. When you are reporting a class name, this parameter should be NIL.
- The fieldType parameter is an integer containing the value corresponding to the type of data that you are reporting. MacApp has defined the following constants to make filling in this parameter easy: bInteger, bHexInteger, bLongInt, bHexLongInt, bString, bBoolean, bChar, bPointer, bHandle, bPoint, bRect, bObject, bByte, bCmdNumber, bClass, bOSType, bWindowPtr, bControlHandle, bTEHandle, bLowByte, bHighByte, bPattern, bFixed, bRgnHandle, bRGBColor, bTitle, bGrafPtr, bSTyle, bVCoordinate, bVPoint, bVRect, bFontName.

The GetInspectorName method

Hex addresses of object instances are displayed in the upper-right pane of the Inspector window. If you want your objects to display more than their instances' names, you need to override the GetInspectorName method.

The GetInspectorName method returns a string in its VAR parameter. This string will be displayed after the hex address in the upper-right pane of the Inspector window. GetInspector name is declared as

```
PROCEDURE TObject.GetInspectorName (VAR inspectorName:  Str255);
```

As an example, the class TWindow uses GetInspectorName to return the window instance's title:

```
{$IFC qInspector}
{$IFC MAInspector}
PROCEDURE TWindow.GetInspectorName (VAR inspectorName:  Str255);
OVERRIDE;
BEGIN
      { Make sure that a window exists }
      IF fWMgrWindow <> NIL THEN
            { If it does, return its name. }
            GetWTitle(fMgrWindow, inspectorName)
END;
{$ENDC}
```

The Interactive Debugger

The MacApp Interactive Debugger is a high-level command-line–operated debugger available to MacApp applications that have been compiled with debugging on. It operates in the Debug window, and is displayed when your application is in debug mode.

When you are in debug mode, the Interactive Debugger prompts you with a list of the one-letter commands it supports.

Entering debug mode

There are several ways you can enter debug mode. One is to select the Enter MacApp Debugger menu item from the Debug menu. You can also type command-grave (⌘-`), which to MacApp is exactly the same as using the mouse to choose the menu item . (The ` key is at the upper left corner on all keyboards except the standard ADB keyboard. You will find the ` key next to the command key at the lower left of a standard ADB keyboard.)

Another way to enter debug mode is to press and hold the attention keys, Shift-Option-Command. This drops you into the debugger on procedure boundaries and may be thought of as the quickest way into the debugger. When you invoke the debugger in this way, you must exercise extreme care when modifying registers and be sure that you return the program counter to the address at which it stopped before you exit the debugger and resume the program.

You enter debug mode at the next {$D++} program point and the Debug window is opened if it is not already on the screeen. The Debugger prints "Stopped at" followed by the name of the routine containing the program point.

◆ *Note:* When running your application under Multifinder, you should be aware that the Shift, Option, and Command keys may not generate user events as other keystrokes would because, strictly speaking, they are modifier keys. Thus, Multifinder would not recognize typing the attention keys as an event and would not pass the keystrokes on in the target chain. If you should experience this problem, you can enter the debugger via the menu item described above, or you can pass the attention keys on in the target chain by causing another event to occur while you are holding them down; for example, clicking or moving the mouse or typing another key .

In addition, debug mode is automatically entered at the next {$D++} program point when

- a breakpoint has been set at program points with this point's identifier (see "The Breakpoint command (B)" later in this chapter), or
- the Interactive Debugger's Single Step command is in effect (see "The Single Step Command (Space Bar)" later in this chapter).

Also, debug mode is entered automatically when

- the procedure ProgramBreak is called in the program, or
- SysErr (the operating system trap that usually displays a "bomb" alert box) has been called. In this case, the program may not be able to continue, and you may be able to use the Debugger to examine the circumstances of the error. (Some 68000 exceptions are included in this group.) At this point, a low-level debugger such as MacsBug is often handy.

For this last group, the debugger can be entered at any point, and not just at {$D++} program points.

Regardless of how you may have entered the debugger, the Debug menu is highlighted to give you a visual cue that you are in debug mode. When in debug mode, you can still use the mouse to pull down menus and see what commands were enabled (drawn in black) or disabled (drawn in grey), but most user commands will not be processed.

The one exception to the above is the Apple menu; all items in the Apple menu are enabled while in debug mode. This is so you can use Multifinder to access another application while debugging your code; for instance, you might want to use the MPW shell to look at source code while you are stopped in the debugger. You may also open desk accessories such as the MacApp Browser from your Apple menu while you are stopped in debug mode.

What the Interactive Debugger prints when it starts

When you enter debug mode, the Interactive Debugger prints a line of text that identifies the routine that was executing, the current value of SELF, and some information about the circumstances under which debug mode was entered. These messages are explained below.

- *Note:* When the debugger prints the messages described below, the placeholders *RoutineName* and *SegmentNumber* shown here will be replaced by the appropriate information. In each case, if the routine is a method, *RoutineName* has the form *ObjectType.Method.*

`BEGIN` *RoutineName* `#`*SegmentNumber*

The routine *RoutineName* has just been entered. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`END` *RoutineName* `#`*SegmentNumber*

The routine *RoutineName* is about to return. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`EXIT` *RoutineName* `#`*SegmentNumber*

An `EXIT` statement in routine *RoutineName* is about to execute. This could be in response to the attention keys, a breakpoint, or the Single Step command.

`BREAK` *RoutineName* `#`*SegmentNumber*

The MacApp global debugging procedure was called from routine *RoutineName.*

`SYSER` *RoutineName* `#`*SegmentNumber*

A 68000 exception (such as divide-by-zero) or a SysError error has occurred.

Using the Debug window in debug mode

The Debug window is a standard Macintosh window with scroll bars, a size box, and a zoom box. You may also manipulate it with these commands:

- `WF` brings the Debug window in front of any other windows.
- `WB` moves the Debug window behind any other windows.
- The Command-Backspace key combination makes the window scroll up.
- The Command-Return key combination makes the window scroll down.

Also, the arrow keys are now supported in the debugger. If you have an extended keyboard, you may also use the Home, End, Page Up and Page Down keys. There is no way to scroll horizontally in debug mode.

Interactive Debugger commands

You can enter Interactive Debugger commands only when in debug mode with the "Command" prompt displayed. Each command is a single character, generally the first letter of the command name.

Some of the commands require parameters. In those cases, a message and another prompt are printed after you enter the command. When entering parameters in response to any prompt in the debugger, terminate your input with Return. An incorrect or inappropriate response will return you either to the previous prompt or to the "Command" prompt, depending on whether or not the particular command you are entering takes parameters.

The rest of this section describes the use of each command. The character you type for each command is given after its command name.

The Help/Status command (?)

The Help/Status command (?) prints some status information followed by the list of Interactive Debugger commands. The status information includes

- the value of A5 and thePort
- whether trace mode is on (see "The Trace command (T)" later in this chapter)
- whether a breakpoint is set and, if so, where
- the identifier and segment for the current {$D++} program point
- whether performance monitoring is turned on (see Chapter 14, "Performance-Measurement Tools," in the *Macintosh Programmer's Workshop 3.0 Reference*)

If you are using an extended keyboard, you can press the Help key as an alternative to entering this command.

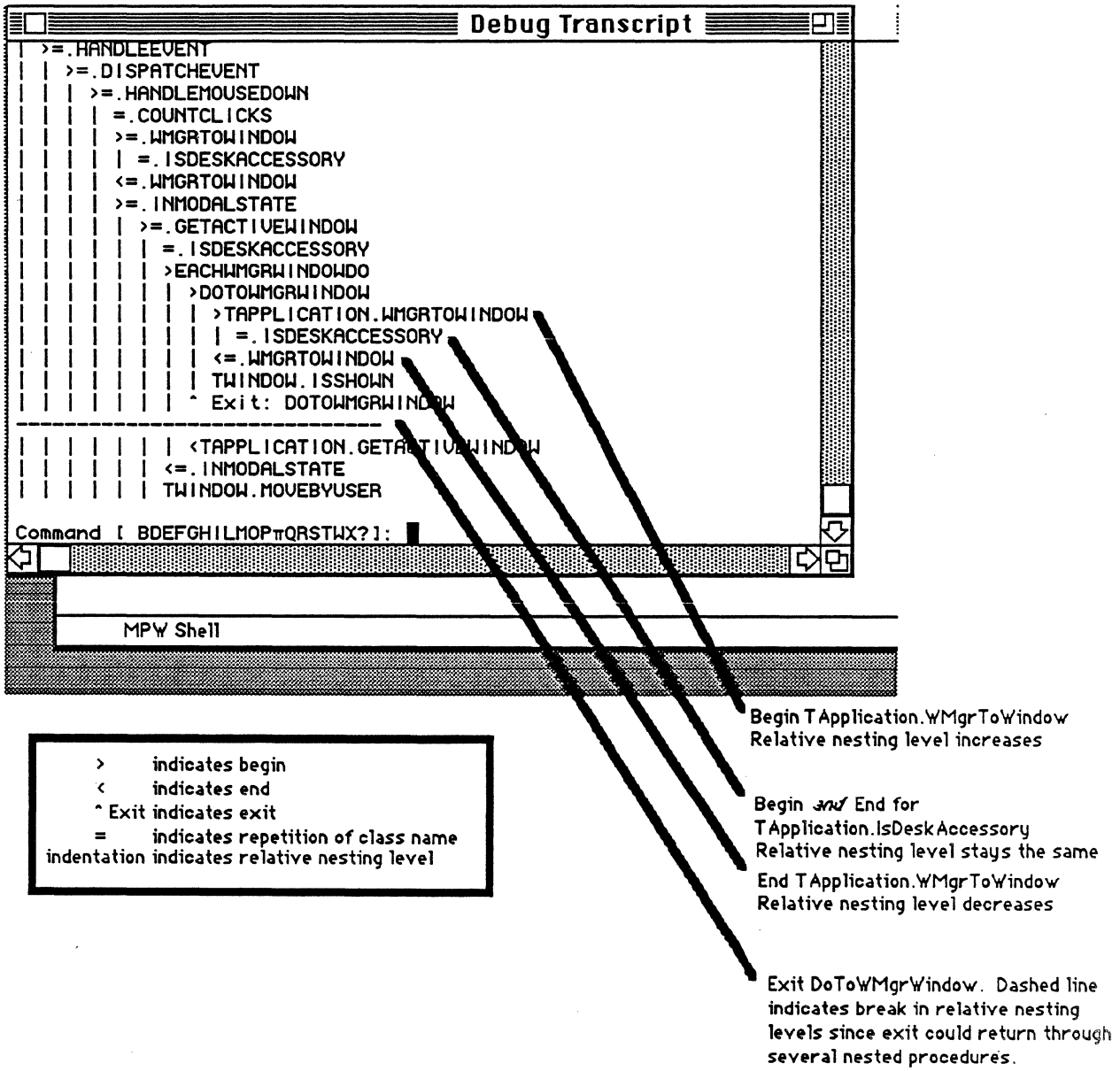
The Recent History command (R)

The Interactive Debugger logs the 63 most recent {$D++} program points that the processor has encountered, whether or not the program stopped at them.

The Recent History command lists all the saved {$D++} program points in the Debug Transcript. The current {$D++} program point is at the bottom of the window, the preceding point is above that, and so on. The oldest recorded {$D++} program point is at the top of the transcript.

A typical program counter transcript as displayed by the Recent History command is shown in Figure 7-2. The greater-than sign (>) indicates the beginning of a method or routine, and the less-than sign (>) indicates the end of a method or routine. As a method calls other methods in the handling of an event, the relative nesting level of the command object in the class hierarchy is indicated by the indentation of the method name as it appears in the Debug Transcript window. The pipe symbol (|) acts like a crude ruler; it has no significance other than to provide further visual cues about the relative nesting level of the methods, routines, and procedures that appear in this window.

■ **Figure 7-2** Output from the Recent History command



In the example shown in Figure 7-2,you can see TEvtHandler.DoToWMgrWindow call the method TApplication.WMgrToWindow. This event is represented on the twelfth and thirteenth lines of the Recent History display in the Debug Transcript window. You can tell that this is the first time we have called a method of class TApplication because the class name, TApplication, is spelled out in the method name. As long as TApplication's methods are in control, listings of subsequent calls to methods of TApplication will have an equal sign (=) in place of `TApplication`. The equal sign is a way of saving space in the Debug window while giving a visual indication that the class name is the same as in the previous line. (Incidentally, there is no visual indication in Figure 7-2 that DoToWMgrWindow is a method of TEvtHandler; were you to scroll the display up further, you would see the class name listed.)

The next line shows both the begin *and* end of a call to TApplication.IsDeskAccessory. You can see this by virtue of the fact that TApplication.WMgrToWindow appears on the lines immediately preceding and following the call to TApplication.IsDeskAccessory. A further clue is that the second listing of WMgrToWindow is marked by the less-than sign (<), indicating that the relative nesting level has decreased. This is because IsDeskAccessory has been handled and the TApplication object is relinquishing control to TWindow.IsShown, listed immediately following `=.WMgrToWindow`.

On the next line we exit TEvtHandler.DoToWMgrWindow, as indicated by the notation `^Exit: DoToWMgrWindow`. The dashed line that follows is a reminder that relative nesting levels on either side of the Exit could be different because this call could return through several nested procedures.

△ **Important** The Recent History command may not work properly (that is, you may get an incorrect name) if automatic segment unloading is enabled (the default). Segments are generally unloaded in the main event loop, so the results of the Recent History command are generally accurate if you have not entered that loop. You can turn off automatic segment unloading by giving the Toggle Flag command (x) followed by a `u`. △

The Parameters command (P)

The Parameters command displays the parameters of a procedure frame. Its output is in hexadecimal form.

When you give this command, you are prompted for a stack frame number. (See "The Stack Crawl command (S)" later in this chapter.) You give the stack frame number as a decimal number and press Return. Level 0 is the routine where the program stopped.

The Locals command (L)

The Locals command displays the local variables of a procedure frame. Its output is in hexadecimal as long as the compiler hasn't optimized variables to use registers.

When you give this command, you are prompted for a stack frame number. (See "The Stack Crawl Command (S)" later in this chapter.) You give the stack frame number as a decimal number and press Return. Level 0 is the routine where the program stopped.

The Fields command (F)

The Fields command displays the fields of an object in hexadecimal form. Also, it shows the object's class and all its superclasses.

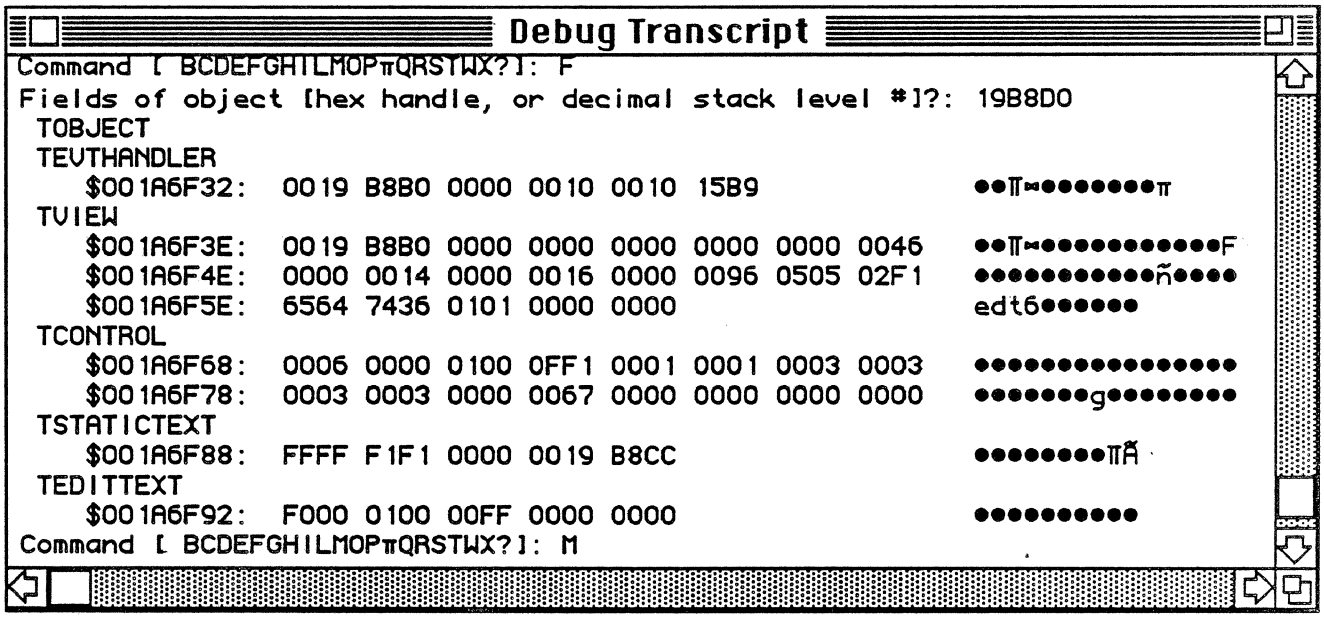
When you give this command, you are prompted for the object you want to know about. You can give an object identifier, a hexadecimal handle, or a decimal stack frame number, and then press Return. The stack frame number is interpreted as referring to SELF for the method in that frame. If the routine in that stack frame is not a method, you are told there is no object at that level. (See "The Stack Crawl command (S)" later in this chapter.)

For certain objects that are referred to by global variables, you can give the global variable identifier. Those global variables are

`gTarget`	`gLastCommand`
`gDocument`	`gApplication`
`gDocList`	`gFreeWindowList`
`gClipView`	`gClipUndoView`
`gPrintHandler`	`gFocusedView`

In the output of the Fields command, the fields of the object are divided into groups determined by the object type that declared the field. That is, inherited fields are listed separately from fields declared for the object's type. The name of each type is printed before the field. Here is an example:

■ **Figure 7-3** Output from the Fields command



In this example, the object occupies locations 1A6F32–1A6F9B. Its type is

```
TEditText = OBJECT(TStaticText)
```

The field declarations define five words (10 bytes) of data, which here occupy locations $1A6F92–$1A6F9B. The immediate superclass, TStaticText, declares fields accounting for five words of data, shown here in locations $1A6F88–1A6F91. The next superclass, TControl, declares fields accounting for 16 words of storage from $1A6F68–$1A6F87. Similarly, the next superclass, TView, declares fields for 21 words in locations $1A6F3E–$1A6F67, and TView's superclass, TEvtHandler, stores six words of data in locations $1A6F32–$1A6F3D. The ultimate superclass, TObject, declares no fields and is always omitted from the Fields command output.

The Display Memory command (D)

The Display Memory command displays the contents of any part of memory.

It prompts for a hexadecimal address and then displays 16 bytes beginning at the specified address. The memory contents are displayed in both hexadecimal and ASCII form.

If you want to see the next 16 bytes, type the Display More command (M) at the next prompt.

The Stack Crawl command (S)

The Stack Crawl command displays a list of the current stack frames. For each stack frame, the command displays the frame number (which can be used in the F, P, L, and I commands), the pointer to the frame in hexadecimal, the name of the procedure or function, and the number of the segment. If the frame belongs to a method, the value and type of SELF are also displayed.

The Display More command (M)

This command can be given only immediately after the Display Memory command or the Stack Crawl command. It displays the next 16 bytes of memory or the next group of stacked procedure calls. See "The Display Memory command (D)" and "The Stack Crawl command (S)" earlier in this chapter.

The Trace command (T)

The Trace command turns trace mode on and off. When trace is on and you give the Go command (to restart your application), the identifier for every {$D++} program point is printed.

The program will slow. Performance is slower if the Debug window is partially hidden by other windows.

Once you restart the program, you are in application mode, and you can stop the program with the attention keys (see "Entering debug mode," earlier in this chapter). Any errors or breakpoints will also stop the program.

- *Note:* You may be able to use the Recent History command to print a trace of the last 63 or fewer {$D++} program points even if trace mode was not on. The display may show fewer lines if it collapses the BEGINs and ENDs of some methods into single lines.

The Single Step command (Space bar)

The Single Step command, given by pressing the Space bar, restarts the program until the next {$D++} program point. The program point is identified. The single stepping will continue for as long as you keep the Space bar held down (unless you have used the Control Panel to turn off auto key repeat.)

The Go command (G)

The Go command ends debug mode and starts the application again. The application continues until the user quits or the Debugger is given some reason to take control again.

The Breakpoint command (B)

The Breakpoint command allows you to set a breakpoint at a {$D++} program point. When you next give the Go command, the program continues until the breakpoint is encountered.

When you give the Breakpoint command, you are prompted for a routine name. Case is insignificant.

If you want to set the breakpoint at a method, you have two choices:

- Give the name of the object type for the method. This will have the form *ObjectType.Method*, where *ObjectType* is the name of the object type that implemented the method. (Note that you cannot give an object-reference variable identifier.) The break will occur only when that particular method is invoked. (That method is invoked either from an object of the type named or from an object of a subclass, if the method named is not overridden by that particular subclass or any intervening subclasses.)
- Give the method name without a qualifying object type. The break will then occur on any method (or ordinary routine) with that name, regardless of what object type implemented it.

The name you give is not checked to verify that it actually exists.

You can set a breakpoint at any routine, regardless of whether it is currently in memory, but note that the break will not occur unless the named routine contains a {$D++} program point.

The Clear Break command (C)

The Clear Break command clears the specified breakpoint. It works in the same manner as the Set Break command.

The Output Redirect command (O)

The Output Redirect command redirects all Debug window output into a file.

When you enter this command, you are prompted for a filename. Enter any filename. (If the file already exists, it must be of type TEXT.) If you do not specify a volume or folder, the output will be written to the default volume in the default folder (that is, the folder containing the application).

If you precede the filename with >>, the Debug window output is appended to the file. (Spaces between >> and the filename are ignored.)

If the file exists and you do not include >>, the contents of the file are erased.

If the file does not exist, it is created and placed in the folder containing your application. Its type is TEXT and its creator is MPS.

To cancel this command, give the command again, and press Return in response to the prompt. The file will be closed.

The Quit command (Q)

The Quit command executes an ExitToShell after you confirm.

The Heap and Stack command (H)

The Heap and Stack command is actually the entry into a group of subcommands. After you give this command, you are prompted with a ? for a subcommand. All the subcommands print information about the heap and the stack.

The Help command (?): Typing ? gives you a list of the available subcommands.

The Set Breakpoint on Procedure Stack Usage command (+): This command allows you to set a breakpoint on stack usage by a single routine.

The stack usage of the routine is computed to be the difference between registers A7 and A6. This value will take into account the local variables declared in the procedure, other local variables used internally by compiler-generated code, and registers that are saved on the stack. It does not take into account any stack usage for passing parameters to routines, stack usage by *Inside Macintosh* routines (including QuickDraw), and stack usage for routines that do not contain a {$D++} program point.

Setting breakpoints on total stack depth and individual procedure usage will help you gauge how much stack space your application requires and to identify routines that have excessively high stack usage. (These may have several string parameters that are passed by value rather than by reference.)

The Set Breakpoint on Total Stack Usage command (B): This command allows you to set a breakpoint on total stack usage. Total stack usage is the difference between register A7 and the bottom of the heap.

Setting breakpoints on total stack depth and individual procedure usage will help you gauge how much stack space your application requires and identify routines that have excessively high stack usage. (These may have several string parameters that are passed by value rather than by reference.)

The Reset Maximum Stack Depth command (D): This command allows you to reset the maximum stack usage value maintained by MacApp to determine if a stack usage breakpoint has been reached. Typing `D` resets that value.

The Show Heap/Stack Information command (H): When you enter this command,the debugger displays a list of available subcommands. Using these commands you can retrieve information about the heap and the stack, such as the values of total stack usage, stack space used by procedures or by resources, maximum stack space used, available stack space, free master pointers, segments loaded, and available free memory. It also shows reserved space for code, low memory, and temporary allocation and indicates amounts of locked, unlocked, and purgeable space.

The Show/Set Heap Reserve command (R): This command shows code reserve and low space reserve in the heap, and then prompts you for a new value for each. If you do not wish to change a value, enter `-1`; otherwise enter a decimal number for the new value in Kbytes. After accepting your new values, you are asked if you want to reset maximum resource usage. After you confirm or deny, all entered values are set and the current values, reflecting your changes, are displayed.

The Print MaxMem command (M): This command prints the value of MaxMem. See *Inside Macintosh* for information.

The List Loaded Segments command (S): This command lists all segments currently loaded. An asterisk (*) next to the segment number indicates that the segment is resident.

The Toggle Flag command (X)

The Toggle Flag command allows you to toggle a number of flags that govern the behavior of the Interactive Debugger.

After you give this command, you are prompted with a list of legal flags, with the current status of each reflected by the list. If the flag appears in uppercase, it is enabled; lowercase indicates that it is disabled. Typing the letter for one of the flags toggles the flag's value. You must set the values of toggle flags one at a time.

You can add new items to the list of flags by calling the MacApp global debugging procedure DebugFlag, found in UDebug. The interface is

```
PROCEDURE DebugFlag(flagAddr: PBoolean; flagChar: CHAR;
            theActionProc: ProcPtr; {CONST} flagDesc: StringPtr);
```

You pass this procedure a pointer to a Boolean variable, a character (used to toggle the flag), and a short description. Only twenty flags are allowed. If you do add flags, be careful not to assign the same letter to two different flags.

The current flags are listed below.

The Discipline Method Calls flag (D): When this flag is on, the debugger checks to see if the receiver of the call is a valid object. If it is not, the debugger breaks in and displays information on possible problems, such as uninitialized instance variables or NIL pointers. If you continue to execute code by typing G, a failure will be signalled.

The Report Memory Management Information flag (M): When this flag is on and the number of master pointers changes, the old and new numbers are printed in the Debug window. When the Break flag is also on, the program is stopped, and you enter debug mode.

The Report Segments flag (S): When this flag is on, a routine name is printed in the Debug window when a segment is loaded. Usually this is the routine that caused the segment to be loaded (unless that routine has no {$D++} program point). When the Break flag is also on, the program is stopped, and you enter debug mode.

The Break flag (B): When this flag is on and the Report Segment flag (S) or the Report Memory Management flag (M) is also on, a break is generated whenever M or S causes something to be printed in the Debug window.

The Automatic Segment Unloading flag (U): This flag, when on, turns off automatic segment loading by the UnloadAllSegments routine. You might want to try this flag if you find that your program is mysteriously jumping off into an unexpected routine; you may have saved a pointer to a routine in a segment that MacApp unloaded.

The Ask About Allocation flag (A): When this flag is on and an object is created, the name of the routine and the type of object are printed in the Debug window. Debug mode is entered, and you are given a chance to replace the object reference with NIL. This is used to check handling of memory management.

The Ask About Failures flag (F): When this flag is on, every call to FailOSErr, FailResError, and FailMemError will give you a chance to enter an error code and make your application fail. This allows you to check that you recover properly from these kinds of errors.

The Report Menu Commands flag (C): When this flag is on and the user chooses a menu command, the command number is printed in the Debug window.

The Report Events flag (E): When this flag is on, details of any events (except Debug window events) are printed in the Debug window.

The Report Resource Usage flag (R): This flag checks to see if the total size of the currently loaded resources exceeds the maximum. If so, the new maximum is set. If gRsrcReport is True, then the new maximum is reported in the debugger window.

The Intense Debugging flag (I): The Intense Debugging flag toggles the value of the global variable gIntenseDebugging. By enclosing code in an IF structure beginning with the statement

```
IF gIntenseDebugging THEN
```

this code will be included only when gIntenseDebugging is TRUE.

The Debug Printing flag (P): When this flag is on and the user does any UPrinting operation, information about printing is displayed in the Debug window.

The Experimenting flag (X): The Experimenting flag toggles the value of the global variable gExperimenting. By enclosing experimental code in an IF structure beginning with the statement

```
IF gExperimenting THEN
```

you can make features work only when gExperimenting is TRUE.

The Inspect Object command (I)

This command allows you to inspect an object.

When you give this command, you are prompted for the identifier of the object you want to inspect. You can identify it with a hexadecimal handle, an object name, or a decimal stack frame number. If you give a stack frame number, the request is interpreted as being for the object used to call the method at that level. (If it is not a method, you are told there is no object at that level.)

For certain objects that are referred to by global variables, you can give the global variable identifier. Those global variables are

```
gTarget
gLastCommand
gActiveWindow
gApplication
gDocList
gFreeWindowList
gFocusedView
gClipView
gClipUndoView
gPrintHandler
```

This command then calls the Inspect method for the given object. Since the inherited Inspect method will call the object's Fields method, you do not need to override the Inspect method for your objects if you have created a Fields method for them.

- *Note:* `gFrontWindow` has been replaced in MacApp 2.0 by `gActiveWindow`. You can define your own global variable to the active document by inspecting the results of `getActiveDocument`, which returns the value of the handle to the document that the active window belongs to. You can then pass this handle to `DebugGlobalHandle`, along with an action proc and a global variable you define. You may then use this global to refer to the active window.

The Enter MacsBug command (E)

When you give this command, MacApp enters MacsBug. You can return to the MacApp debugger by giving MacsBug the Go command (G).

This command is equivalent to pressing the interrupt button on the Macintosh programmer's switch.

Using MacsBug with MacApp

Although the MacApp debugging facilities are powerful, you may sometimes need to use MacsBug, the lower-level Macintosh debugger. See the *MacsBug 2.0 Reference* for details on using MacsBug; this section has some additional information you may find useful for using MacsBug with MacApp programs. Macsbug is a separate product, available from APDA at the address listed in the Preface of this book.

You can set breakpoints on methods as you do with any routine. If you want to set a breakpoint on all method calls, you need to know something about how method calls are implemented.

Method calls are implemented in different ways, depending on whether the code has been partially or fully optimized using the optimization option method calls of the MPW Linker. Method calls are routed through the method dispatch routine, %_NEWMETHOD. If you want to set a breakpoint on all methods, you can do so by setting a breakpoint on %_NEWMETHOD.

You cannot use MacsBug to break on all method calls if you built the application with the `-NoBottleNeck` option. (The `-NoBottleNeck` option is only necessary when debugging if you are running out of jump table space.)

If you've stopped at a {$D++} program point, you can use a different method to invoke MacsBug in the routine containing the program point:

1. While stopped at the {$D++} program point (which you can reach by setting a MacApp breakpoint or in any other way), invoke MacsBug from the MacApp Interactive Debugger.
2. Give the MacsBug SC (Stack Crawl) command. A line of this form is displayed:

   `SF` *@address* `FR` *fromAddress Method.Object +offset*

   where *address* and *fromAddress* are two addresses, *Method* is the routine containing the {$D++} program point, *Object* is the type that defined that method, and *offset* is the offset into the routine.
3. Add 4 to the *fromAddress* and, using that number, type

   `GT` *address*

   where *address* is *fromAddress* plus 4. This sets a temporary breakpoint just beyond the {$D++} program point.
4. Return to the MacApp Debugger and type `G`. You will enter MacsBug in the routine with the {$D++} program point.

Appendix A **Changes Since MacApp 1.1**

MACAPP 2.0 DIFFERS FROM MACAPP 1.1 in many ways. This appendix is a brief introduction to the architectural differences. Despite these changes, you should still find that you will be able to convert your old MacApp applications into MacApp 2.0 very quickly. Once you know how MacApp 2.0 works, you should be able to convert even rather large applications in less than a day. Detailed instructions on converting MacApp 1.0 and 1.1 applications to MacApp 2.0 are given in the *MacApp 2.0 Cookbook.* ■

Changes to the architecture

The architecture, or overall design, of MacApp has changed in a great many ways.

- **Views.** The view architecture has been completely overhauled. All classes responsible for displaying images on the screen are now subclasses of TView. The TFrame class no longer exists.

  Views now offer a larger coordinate system: Two billion pixels both horizontally and vertically.
- **TextEdit.** The new TextEdit supports views that allow multiple fonts, styles, and sizes.
- **Dialog boxes.** MacApp 2.0 has a whole new design for implementing dialog boxes, allowing you to create dynamic, sophisticated dialogs with smart controls that you can customize.
- **Grids.** MacApp 2.0 has a new unit containing code that will allow you to manipulate data grids, as in spreadsheet applications.
- **Inspector.** The debugging capabilities now include an inspector, which you can use to examine the fields of any instance while the application is running.
- **AppleTalk.** MacApp no longer contains a special AppleTalk® support unit.
- **MultiFinder.** New support for MultiFinder features.

These changes are described more fully below.

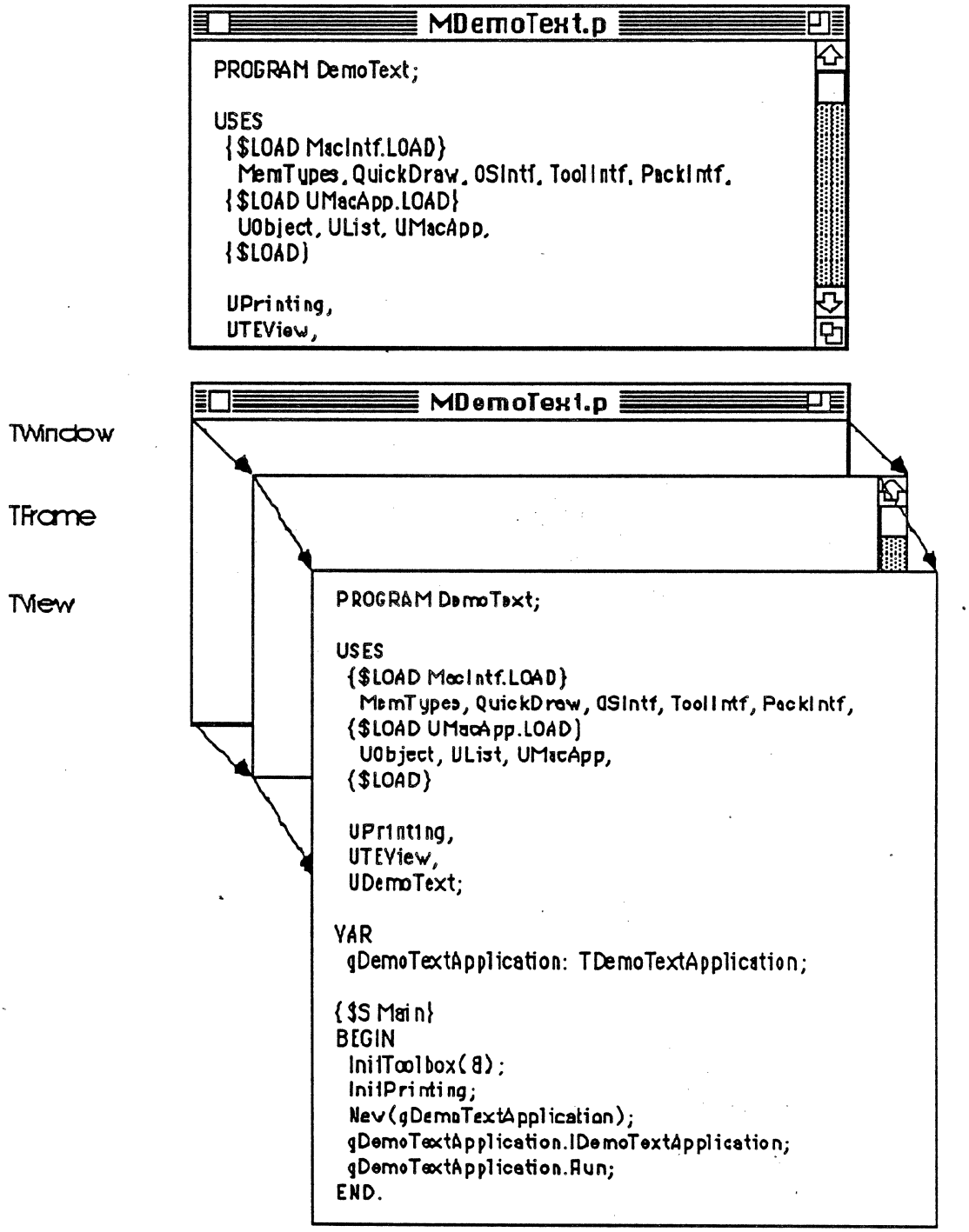
Views

MacApp 2.0 offers two major improvements over MacApp 1.1 in the display architecture: it greatly simplifies the architecture and it provides a larger coordinate system.

The old display architecture

MacApp 1.1 provided three basic classes for displaying information on the screen: TWindow, TFrame, and TView. TWindow provided a Window Manager window, TFrame managed the scroll bars and the coordinate translation for scrolling, and TView was responsible for drawing the contents of the window and for handling mouse commands inside the view. This architecture made it difficult to nest views and frustrating to provide dialog boxes.

■ **Figure A-1** The display architecture from MacApp 1.1



The new display architecture

Whereas the old architecture had two separate lineages (TFrame and TView) that used different schemes for nesting views, the new 2.0 architecture has only one: TView. All display classes descend from TView and they share the following abilities:

- nesting
- focusing
- drawing
- handling events

A view is still an imaginary construct that displays information. On the screen, views are rectangular. One view may be inside another view; also, views may overlap.

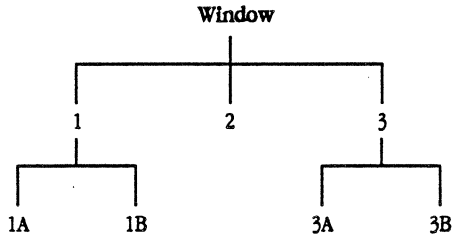
Nesting. Each instance of the TView class (or its subclasses) may be inside another view, called the superview; and it may have any number of other views, called subviews, inside it. This relationship between superviews and subviews forms a tree. Note that these terms are relative: a single view may be both a superview in relation to one view and a subview in relation to another.

On the screen, nesting is a matter of layering the views. Superviews are at the bottom. A subview always covers the superview: you cannot usually see the superview through one of its subviews. However, superviews do clip subviews. Only that portion of the subview that intersects with the superview is visible.

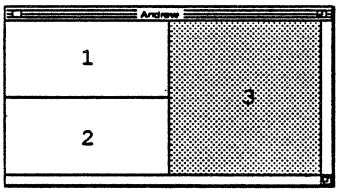
If a superview has more than one subview, the subviews are also arranged in layers. If two subviews overlap, one of them will be on the top and the other will be on the bottom.

In Figure A-2, there are eight views arranged into a tree. View A has no superview, but it does have three subviews. Note how the layering appears on the screen, and how superviews clip their subviews.

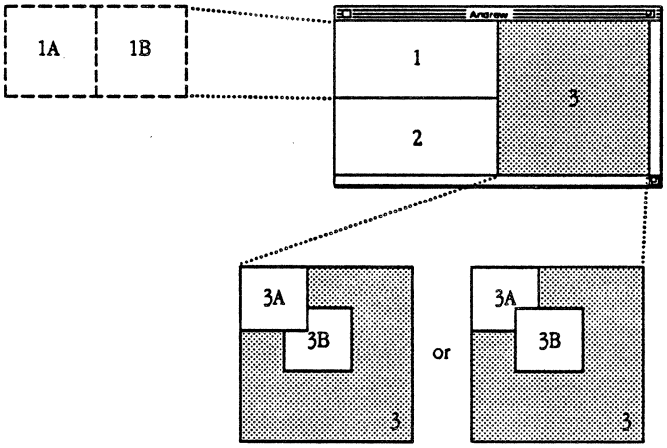
■ **Figure A-2** Tree of views



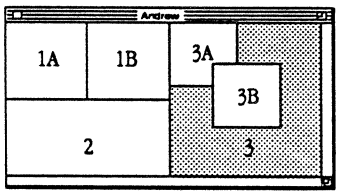
Window has 3 subviews



Views 1 and 3 each have 2 subviews.



Window and its subviews as the user sees them.



Focusing. The concept of focusing has not changed. Each view has its own local coordinate system. Focusing translates between the different coordinate systems, and prepares the view so that your drawing will appear in the correct place on the screen.

Drawing. Each instance of the view class has a Draw method, which you are expected to override. Whenever the view draws itself, it tells its subviews to draw themselves. The process continues iteratively until all the subviews are drawn.

Handling events. TView is still a subclass of TEvtHandler, and thus it inherits the ability to handle events including keystrokes, menu commands, and other mouse clicks from the user. Because views can be nested, MacApp needs an algorithm to determine which particular view instance will handle a particular mouse command. Based on the metaphor of subviews layered on top of superviews (and on top of their sibling views), the algorithm is simple: The command is handled by the topmost view at the location of the mouse click. Mouse clicks are handled by the method HandleMouseDown.

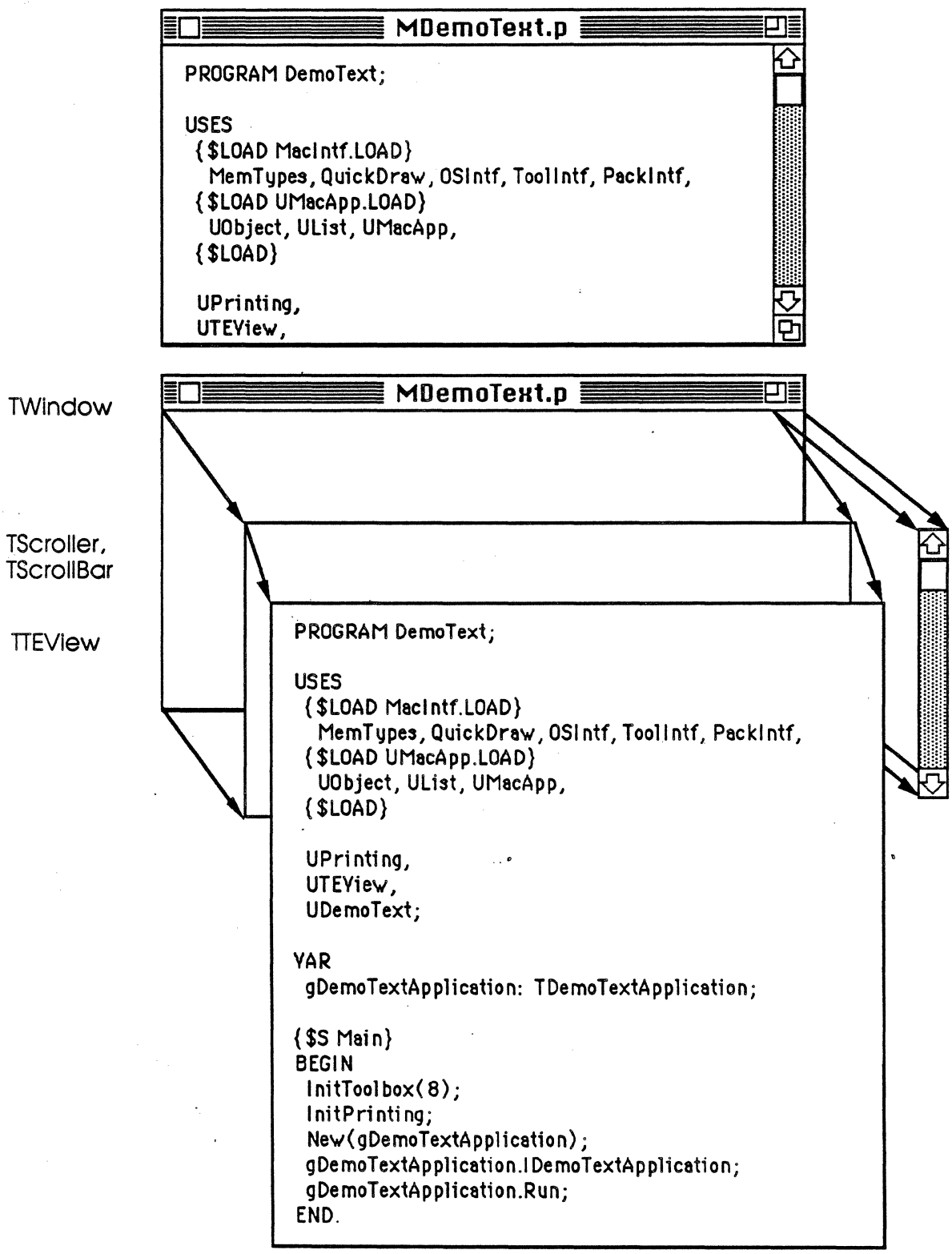
The most important classes in the new architecture

The major players in the new MacApp 2.0 display architecture are TWindow, TSScrollBar, TScroller, and specialized content views. These are all subclasses of TView.

- TWindow displays the window frame and size box and handles clicks in the title bar, zoom box, and size box.
- TSScrollBar displays the scroll bars and communicates with TScroller to carry out the user's commands.
- TScroller does the actual calculations for scrolling.
- Specialized content views display whatever you want them to. MacApp provides several of these classes: TEView for displaying text; TGridView for displaying spreadsheets and other information that fits well into grids; and a number of others defined in the UDialog unit. To display other sorts of information you must create your own subclasses of TView.

Everything that appears on the screen (except menus) appears inside a window. The window then becomes the root of the view tree—it has no superview. If some portion of the contents can be scrolled, the window instance will have a subview that is a scroller instance and perhaps one or two instances of TSScrollBar (vertical or horizontal or both). Anything that can be scrolled should be a subview of the scroller. For an example, see Figure A-3.

■ **Figure A-3** View tree for a MacApp window



Large coordinates

MacApp 2.0 provides a large coordinate system so you can implement bigger views. The old MacApp was limited to QuickDraw's 30,000 by 30,000 pixels. QuickDraw hasn't changed that limitation, but MacApp 2.0 provides types to store these large 32-bit coordinates: VCoordinate, VPoint, and VRect. With these types, you can store views up to two billion pixels by two billion pixels.

All drawing is still done using QuickDraw, in 16-bit coordinates. If you want to use 32-bit coordinates, the burden is on you to convert between 32-bit and 16-bit coordinates, using routines and methods provided by MacApp's UViewCoords unit and the TView class.

Text

The TTEView optional building block (in the UTEView unit) has been enhanced to support the new display architecture and to allow your TEViews to have multiple fonts, styles, and sizes. The building block defines the same classes that it used to, but it has additional methods to deal with the different commands; and the old methods have been changed to record and deal with information related to font, style, and size.

TTEView is still based on TextEdit in ROM, so it has the same limitations as TextEdit, namely a maximum of 32,000 characters per view and a maximum height and width of 30,767 pixels each.

Dialog boxes

MacApp no longer makes use of the ROM's Dialog Manager. Instead, dialog boxes are now constructed the same way as any other MacApp window—by constructing a hierarchy of views. The new UDialog unit simply provides a set of predefined view classes that can be used in *any* window. One class, TDialogView, can provide modal dialog behavior such as tabbing between text fields and implementing default and cancel buttons. Here are some of the elements defined by MacApp which were traditionally found in dialog boxes:

- static text
- editable text
- icons
- pictures
- check boxes

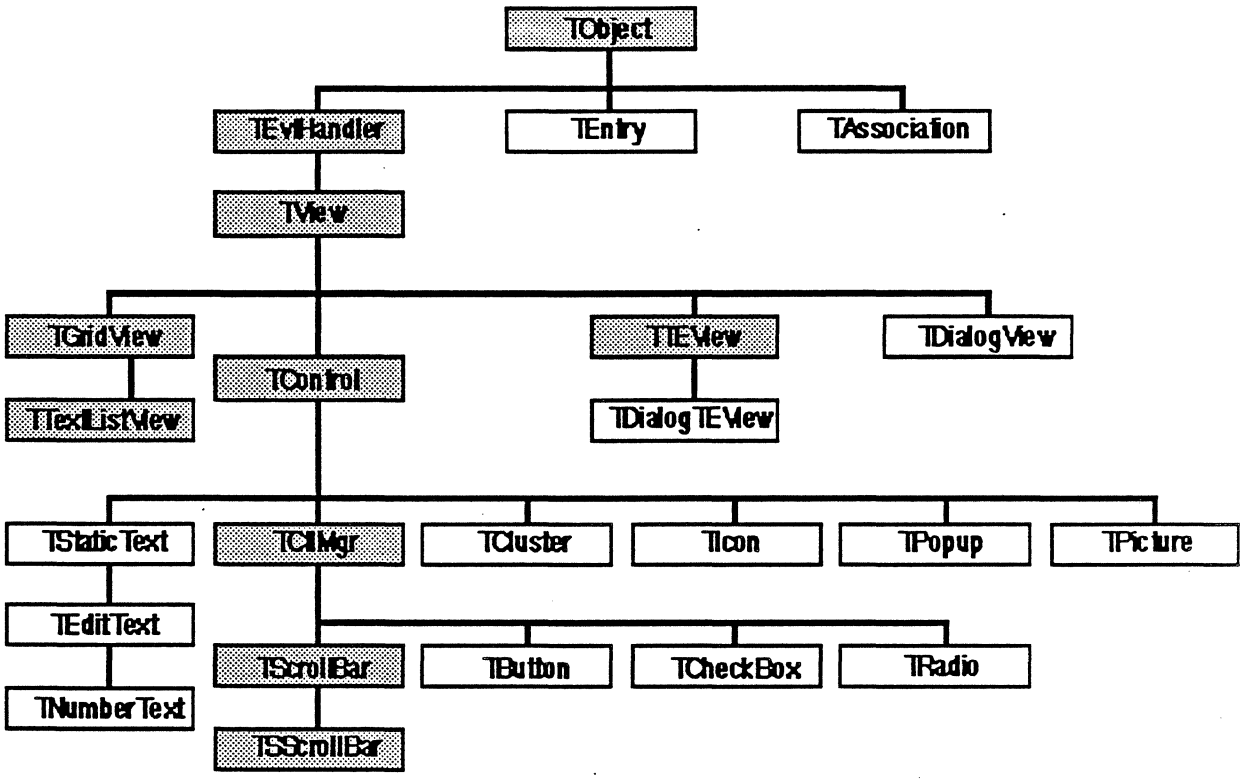
- radio buttons
- plain buttons
- pop-up menus
- scrollable lists
- clusters, which define dependencies among the controls

The two major advantages this new scheme gives you are that (1) you can use any of the classes of UDialog in any window and (2) there is no need to distinguish between "windows" and "dialog boxes" as there was in MacApp 1.1 and as there continues to be when using the Toolbox directly.

Note that the new display architecture blurs the distinction between windows and dialog boxes. Any window can be modal or modeless. To create the illusion of a dialog box, you will use the UDialog unit to create either a modal or a modeless dialog with a variety of controls.

MacApp defines a large number of classes associated with dialog boxes, most of which can be found in the UDialog unit:

■ **Figure A-4** The classes defined in UDialog



TEntry. Instances are metaphorical text items, to be used as entries in a dictionary, managed by an instance of TAssociation. Instances of TEntry are used mainly for substituting text in dialog boxes.

TAssociation. Instances manage lists of instances of TEntry.

TDialogView. Instances duplicate the function of a dialog window. An instance of TDialogView acts as the topmost view in a template-driven dialog box and in some ways replicates the function of the ROM Dialog Manager.

TButton. Instances implement a Control Manager button.

TCheckBox. Instances implement a Control Manager check-box.

TRadio. Instances implement a Control Manager radio button.

TScrollBar. Instances implement a scroll bar, which is one of the more popular types of dials available in Macintosh dialogs.

TCluster. Instances implement a container for a cluster of radio buttons or other objects. The cluster can have a label. Clusters are used to group radio buttons and to localize parts of a window.

TIcon. Instances implement an icon which can serve as a static image, or work as a button.

TPicture. Instances implement a picture, which can serve as a static image or work as a button.

TPopup. Instances implement a pop-up menu.

TStaticText. Instances implement a text item, which can serve as a static image, or work as a button.

TEditText. Instances implement a simple editable text item. When the item needs to be edited, the parent DialogView places a floating TEView over the top of the view.

TNumberText. This is a specialized version of TEditText which accepts only integer numbers.

TDialogTEView. This class is intended for use with TDialogView and not intended to be used for your views.

Grids

MacApp 2.0 offers a new optional building block, UGridViews, which implements several classes that make it easier to display and manipulate gridlike information, such as spreadsheets. The widths and heights of the individual cells can be adjusted dynamically in response to the user's input. GridViews do not dictate how the data is displayed; the representation of that data is completely up to you.

The TGridView unit includes definitions for the following classes:

TGridView. Instances display a one- or two-dimensional grid of cells. The cells can contain anything you wish to draw: text, PICTs, other views, etc.

TCellSelectCommand. A command class that implements the command that selects a cell.

TRowSelectCommand. A command class that implements the command that selects a row divider.

TColumnSelectCommand. A command class that implements the command that selects a column divider.

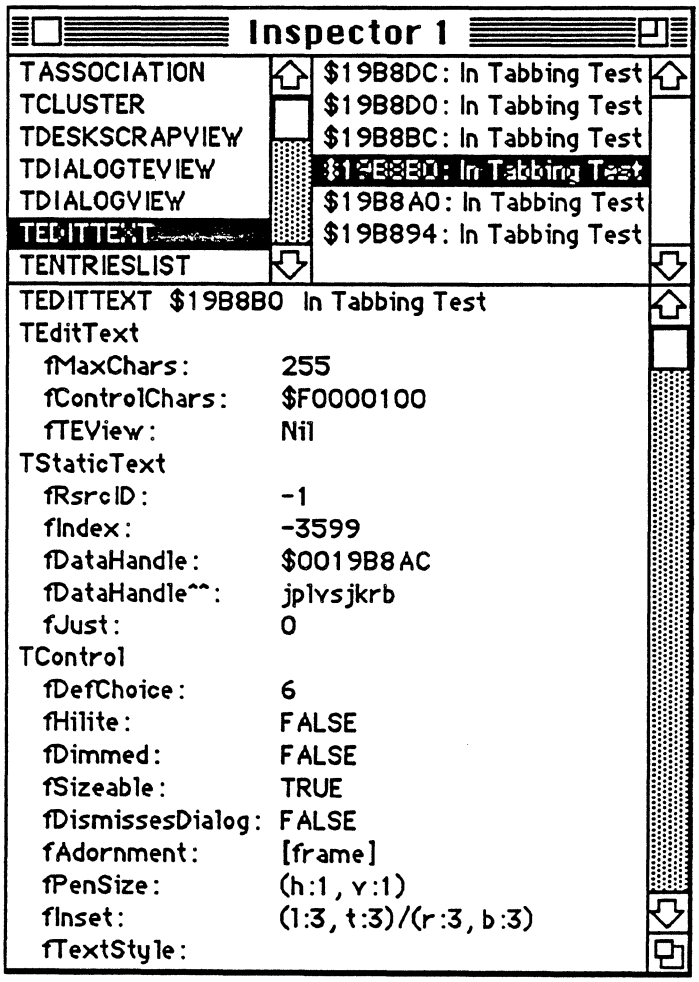
TTextGridView. Instances display a one- or two-dimensional grid of cells that can display only text.

TTextListView. Instances display a one-dimensional list of cells that can display only text.

Inspector

MacApp 2.0 features an Inspector for examining the contents of any object's fields. When you are running a debug version of your application, you can select the New Inspector Window command in the Debug menu. See Figure A-5 for an example of an Inspector window.

■ **Figure A-5** An Inspector window



Using the Inspector, you can select any object and then examine the contents of its fields, as described in Chapter 8, "MacApp Debugging Facilities."

AppleTalk

MacApp 2.0 contains no UAppleTalk unit. Though some Macintosh application programs have some AppleTalk functionality in their applications, it remains to be seen whether there is enough in common among all these programs to warrant an optional AppleTalk building block.

MultiFinder support

While programs built with MacApp 1.1 work under MultiFinder, they don't really take advantage of it. MacApp 2.0 makes use of the WaitNextEvent trap and provides a mechanism for identifying the cursor region and length of time before MultiFinder needs to wake up the application. Also, MacApp 2.0 allows for read-only files.

Index

+ Command 177

A

About command 98, 115
abstract class 33
activate event 111-112
Activate method 118
activating windows 89, 111, 118
alien events 110-112, 125
allocating memory 69
Allow Trace During Idle command 161
Allow Trace Of Menu Setups command 161
ancestor class 27-31
ancestor, immediate 31
ancestry in MacApp 79-85
AppleTalk 185, 196
Application.MAMake file 140
application mode 159
applications. *See also* programs
  building 3, 133-137
  debugging 153
  one-file configuration 137
  production version 155
  structure of MacApp 150-152
  under MultiFinder 167
AppName variable 140
`-autoBuild` option 136
A/UX 7

B

Basic Definitions makefile 135
B command 176, 177
bit images 88
bit maps 88
border, flashing 126
borrowing code 5
Breakpoint command (B) 176
breakpoints, setting 177-178
Browser, MacApp 167
Build Rules and Dependencies makefile 135
BuildBlockObjs variable 140
BuildingBlockIntf variable 140
buzzword menus 149

C

calling instance 19, 31, 59, 61
CASE statement 109
C command 176
chains of responsibility 114-126
class definitions 51-54
classes 13-16
  abstract 33, 82
  creating 34
  with dialog boxes 192-194
  event-handling 83
  hierarchy of 27-31
  inheritance with 27-31
  organization of 79-85
Cleanup routine 107
cleanup tasks 69
Clear Break command (C) 176
click chain 114, 115, 123-124
close box 93, 113
`cmnu` resources 148, 149
cohandler chain 115, 125-126
columnar data 96
command chain 114, 115-122
command continuation character 141
Command-key combinations 98, 111, 113-114
commands 83, 98
  Debug menu 160-62
  executing 98-100
  implementing users' 87
  Interactive Debugger 169-181
  menu 6, 114, 119
  mouse 189
  view 115
  window 115
compiler options, setting 144-146
compiler switches 152
compiler variables 156
compile-time variables 144
content views 95, 189
control, flow of 17-26
coordinate system 191
customization 5, 28, 43
Cut command 115

D

$D switches 157-158
data
  columnar 96
  displaying 84
  storing 86-87
data encapsulation 41
data fields. *See* fields
D command 174, 178
Debug Files folder 139
`-Debug` flag 139, 155
debugger. *See* Interactive Debugger
Debug menu 149, 153, 160-162
debug mode 159, 166-167, 169
`-Debug` option 139, 155
Debug window 153, 159-160
DebugFlag procedure 178
DeRez 149
descendant classes 27-31
descendant, immediate 31
desk accessories 3, 113
device drivers 3
dialog boxes 96, 185, 191-194
Dialog Manager 85
disk event 111-112
disk space, saving 131
DispatchEvent method 107-110
Display Memory command (D) 174
Display More command (M) 175
documents 7, 86-87, 115
Do First Click For This Window command 161
DoHandleEvent method 125
DoIdle method 114, 125
DoKeyCommand method 99-100, 114

"Do" methods 127
DoMenuCommand method 87, 99-100, 114
DoMouseCommand method 99-100, 114
DoSetupMenus method 87, 161
double indirection 56, 57
drawing views 189
Draw methods 95, 112, 189

E

E command 181
editing text 7, 85
Enter MacApp Debugger command 161
Enter MacsBug command (E) 181
errors, handling 6
event-handler fields 86, 87
event handling 105-114, 127
event record 113
events 83
  alien 110-112
  assigning to instances 111-114
  categories of 105, 111
  idle 125

F

F command 173-174
fIdleFreq field 125
field declaration 52-53
fields 12, 13, 15
  customizing 38
  declaring 52
  examining contents of 195
  modifying 72
  referencing 57
  tabbing between text 191
Fields command (F) 173-174
Fields method 164-165
filenames in MacApp 78-79, 132-133
file organization 47-50
files
  implementation 47-50, 78, 152
  Include 148
  interface 47-50, 78
  in MacApp units 78
  naming conventions 78-79, 132-133
  public interface 151-152
Finder 106
fIsModal flag 160
flags 141, 156, 178-180
flow of control 103
fNextHandler field 117-119
focusing views 189
folders 131, 136, 139
FORWARD declaration 151
Free method 69
FUNCTION keyword 53
function method calls 61
function methods 18, 53

G

G command 176
GetInspectorName method 165-166
gHeadCohandler variable 125
global objects 22-23, 61-62
global variables 173, 180
Go command (G) 176
grafPorts 88
graphical menus 150
grids 84, 185, 194
gTarget variable 118

H

HandleAlienEvent method 108, 110
HandleEvent method 107-110
HandleMouseDown method 123, 189
handles 56
hardware requirements xv, 129
H command 177, 178
Heap and Stack command (H) 177
Help command (?) 177
Help/Status command (?) 170

I-J

I command 180-181
idle chain 114, 125
idling 111, 114
immediate ancestor 31
immediate descendant 31
implementation files 47-50, 78, 152
Include files 148
inheritance 27-31, 37-40
inherited methods 34, 68
initialization methods 22-23, 69-71
initializing objects 65-66
Inspect methods 164, 180
Inspect Object command (I) 180-181
Inspector 185
Inspector windows 153, 160, 162-166, 195
InstallCohandler method 125
Installing MacApp 131
instances 13-16, 55-58. *See also* objects
  assigning events to 111-114
  creating 69
  event handling by 98, 114
  freeing 69
  privacy between 41, 72
Interactive Debugger 153, 157, 166-181
interface files 47-50, 78

K

keyboard commands 98, 111
KeyDown event 111-114

L

L command 173
Linker 181
List Loaded Segments command (S) 178
local object 62
Locals command (L) 173

M

{MABuildDefaults} variable 147
MABuild system 133
  command 133, 139
  inputs 135
  modifying default behavior 138
  options 135, 139, 141-143
  outputs 135
  script 133, 135
  tool 133
MacApp
  ancestry in 79-85
  benefits of 7-8
  coordinate system 191
  debugging facilities 153
  filenames in 78-79
  flow of control 103
  installation 131

overview of 106-110
standard Pascal in 79
system requirements 129
2.0 versus 1.1 183
units in 77-78
MacApp Browser 167
MacApp disks 131
MacApp folder 131
Macintosh Programmer's Workshop (MPW) xv
MacsBug 153, 155, 181-182
{MADone} variable 147
{MAFailed} variable 147
main content view 95
main event loop 4, 105
main file 47-50
main procedure 22-24
main program 150
MainEventLoop method 107-110
Make command (MPW) 133
makefiles 135, 139-140, 137-138
Make Front Window Modal command 160
MakeIt file 133
Make tool (MPW) 133
Make variables 140
{MAUserAutoBuild} variable 147
MBAR resources 148, 149
M command 175, 178
member (of class) 36
menu commands 6, 119
Menu Manager 150
MENU resources 148, 149
menus 6, 113, 149, 150
messages, sending 15
method declaration 52-53, 70
method definition 31-32, 52-53, 58-68
methods 13-16
calling 26, 61-66
customizing 37
declaring 52-53
defining 58-68
event-handling 127
function 18, 53, 61
initialization 22-23, 69-71
naming conventions 53
override 32-34, 37-40, 54
procedure 18, 53, 61
syntax for calling 57-58
mntb resources 149
mouse clicks 93, 111-114, 189
MouseDown event 111-114
MouseUp event 111-112
MultiFinder 7, 111, 159, 167, 185
multiple fonts 191

N

NeededSysLibs variable 140
nesting 187
network event 111, 114
New command 98, 115
New Inspector Window command 160
NEW routine 55, 57, 69
NewHandle routine 56
-NoBottleNeck option 181
-NoDebug option 139
Non-Debug Files folder
{No} option specifier 141

O

object classes. *See* classes
object instances. *See* instances
OBJECT keyword 51
object methods, calling 26
object-oriented programming 5, 9
benefits of 42-43
vs. conventional programming 11-12
flow of control in 17-26
Object Pascal 3, 9
defining a class in 51-54
file organization 47-50
syntax summary 73-74
object reference fields 36, 62-63, 88, 115
object reference variables 56-57, 72-73
objects 3, 11-16. *See also* instances
global 22-23
idle 125
initializing 65-66, 69-71
inspecting 180-181
local 62
one-file configuration 138
as records 12-13
O command 176
options
compiler 146
MABuild 141-143
OR condition 109
OtherInterfaces variable 140
OtherLinkFiles variable 140
OtherRezFiles variable 140
OtherRsrcFiles variable 140
OtherSegMappings variable 140
Output Redirect command (O) 176
OVERRIDE keyword 54, 67, 151
override methods 32-34
class definition with 54
defining 67
and inheritance 37-40
limitation 54

P

palettes 90, 95, 117, 123
parameters
in debugger commands 169
in MABuild options 141
with methods 53
Parameters command (P) 172
Pascal 3, 9
Pascal units 47-50, 77-78
Paste command 115
P command 172
+ command 177
pointers, record 55
PollEvent method 107-110
PostHandleEvent method 107-110
PostRez tool 149
printing 7, 159
Print MaxMem command (M) 178
privacy between instances 41, 72
PROCEDURE keyword 53
procedure method calls 61
procedure methods 18, 53
programming a Macintosh 4-5
programs. *See also* applications
building 133-36
format in Object Pascal 47
printing out 170-172
structure of MacApp 150-152
-Pr option 144, 146
public interface file 151-152

Q

Q command 177
qDebug variable 156, 158
? command 170, 177
QuickDraw 88, 159, 191
Quit command 98, 122, 177

R

R command 170, 178
Read statements 160
ReadLn statements 160
Recent History command (R) 170
record handles 56
record pointers 55
record type definitions 12-13
record variables 55
Refresh Front Window command 161
`-RenameFlag` option 144
Reset Maximum Stack Depth command (D) 178
Revert to Saved command 87
Rez files 140, 148-150
ROM 4
`-Run` option 141

S

SADE 139
Save As command 87
Save command 100
SC command 182
S command 175, 178
scroll bars 90, 95-96
scrollers 90, 94-95
seg! resources 148
SELF keyword 58-61
-SeparateObjects option 136, 144
Set Breakpoint on Procedure Stack Usage command (+) 177
Set Breakpoint on Total Stack Usage command (B) 177
shell variables, MPW 133, 135, 140, 147
Show Clipboard command 98
Show Debug Window command 162
Show Heap/Stack Information command (H) 178
Show/Set Heap Reserve command (R) 178
Show Software Version command 162
Single Step command 157, 175
size box 93, 113
`SIZE` resources 148
software requirements xv, 129
Space bar, in debugger 175
Stack Crawl command 175, 182
stepping through a program 175
storing data 86-87
subclasses 27-31
subviews 88-89, 187
superclasses 27-31, 32-34, 51
superviews 88-89, 187
Switcher 111
`-Sym` option 139
SysErr error 167, 168
system event 111-112

T

TApplication class 83, 151
target (application) 138
target chains 115
TAssociation class 82
T command 175
TCommand class 82, 83, 85, 100
TControl class 85, 95-96
TControl instance 112
TDeskScrapView class 85
TDialogView class 85, 191
TDocument class 84, 86-87
`-Te` option 144, 146
TEntry class 82
TEView class 96-97
TEvtHandler class 82, 83, 85
text 7, 96, 159, 191
TextEdit 85, 185
text editor 7
TFrame class 185
TGridView class 84, 90, 96-97
TList class 82, 83, 85
TListView class 84
TObject class 51, 69, 82, 174
Toggle Flag command (X) 178-180
Toolbox 4, 111
Trace command (T) 175
trace mode 161, 175
TRunArray class 82
TScrollbar class 96
TScroller class 84, 90, 94, 96, 189
TSScrollBar class 90, 95-96, 189
TTEView class 85, 191
TView class 84, 88, 89, 90
TWindow.Activate method 118
TWindow class 84, 90, 93, 189

U

UDebug unit 178
UDialog unit 96-97, 155, 191-192
UGridViews unit 194
undoable commands 100
unit public interface file 151-152
units 47-50, 77-78
update event 111-112
UPrinting unit 155
user event 98
UserStartup file 131
USES statement 47
UTEView unit 96, 155, 159, 191
UTranscriptView unit 159

V

variables
- compiler 156
- global 173
- Make 140
- object reference 56-57
- records as 55

`view` resources 148
views 84, 95, 185-189
- activating 89
- commands in 115
- drawing 189
- focusing 189
- hierarchy of 88-89
- main content 95
- nesting 187
- palette 95

W

WaitNextEvent trap 196
`WB` command 169
`WF` command 169
Window Manager 93, 111
windows 6, 84, 89-93
- activating 89, 111, 118

clicks in 113, 123-124
commands in 115
and dialog boxes 192
WriteLn statements 155,159
Write statements 159
-Wt option 144, 146

X-Y

X command 178-180

Z

zoom box 93, 113

THE APPLE PUBLISHING SYSTEM

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh® computers and Microsoft® Word software. Proof and final pages were created on the Apple LaserWriter® printers. Line art was created using Adobe Illustrator™. POSTSCRIPT®, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Apple's corporate font, a condensed version of Garamond. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.